



GENERATING EXECUTABLE PERSISTENT DATA

STORAGE/RETRIEVAL CODE FROM
OBJECT-ORIENTED SPECIFICATIONS

THESIS

Steven R. Buckwalter, Capt, USAF

AFIT/GCS/ENG/00M-02

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

Approved for public release; distribution unlimited

20000815 167

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U. S. Government.

AFIT/GCS/ENG/00M-02

Generating Executable Persistent Data Storage/Retrieval
Code from Object-Oriented Specifications

THESIS

Presented to the Faculty
Department of Electrical Engineering
Graduate School of Engineering and Management
Air Force Institute of Technology
Air University
Air Education and Training Command
In Partial Fulfillment of the Requirements for the
Degree of Master of Science in Computer Science

Steven R. Buckwalter, B.S. Computer Science
Capt, USAF

March, 2000

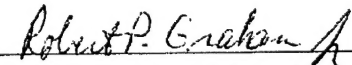
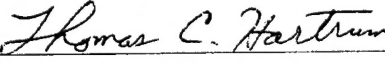
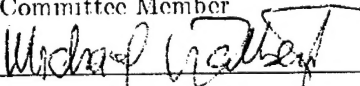
Approved for public release; distribution unlimited

AFIT/GCS/ENG/00M-2

Generating Executable Persistent Data Storage/Retrieval
Code from Object-Oriented Specifications

Steven R. Buckwalter, B.S. Computer Science
Capt, USAF

Approved:

 Maj. Robert P. Graham Committee Chair	<u>3 MAR 2000</u> Date
 Dr. Thomas C. Hartrum Committee Member	<u>3 MAR 2000</u> Date
 Maj. Michael L. Talbert Committed Member	<u>3 MAR 2000</u> Date

Acknowledgements

I wish to thank my thesis advisor, Maj. Graham, for always being available to encourage and advise. He consistently urged me towards my best performance, much as a shepherd leads his flock. I also thank my committee members, Dr. Hartrum and Maj. Talbert, who often provided timely advice and insight. I would also like to thank the rest of the KBSE group their support and for providing a sounding board for ideas.

I owe a debt of gratitude to my lovely wife and best friend, Patricia, and our two children, Benjamin and Daniel. They were a constant source of encouragement and understanding through many late nights and working weekends.

Steven R. Buckwalter

Table of Contents

	Page
Acknowledgements	iii
List of Figures	vii
List of Tables	ix
Abstract	x
 I. Introduction	 1
1.1 Background	2
1.2 Problem	2
1.3 Assessment of Past Effort	3
1.4 Scope	3
1.5 Contributions	3
1.6 Document Structure	4
 II. Background	 5
2.1 Introduction	5
2.2 Background Information on AFITtool	5
2.3 From GOM Through COIL to AWSOME	6
2.4 Transformation Methods Between Object-Oriented and Relational Methodologies	8
2.4.1 Informal Case Tools for Automated Development of DDL	10
2.4.2 Informal Case Tools for Automated Development of DML	12
2.4.3 Formal DML Transformation Tool	12
2.4.4 Conclusion	13

	Page
III. DDL Generation	15
3.1 Mapping Classes to Tables	17
3.1.1 wsAbstract and Class Persistence	17
3.1.2 Attributes	19
3.1.3 wsClassSuperclass and Specifying Inheritance	20
3.1.4 Transformation of Classes	22
3.2 Associations	24
3.2.1 Representation of Associations in AWSOME	24
3.2.2 Analysis of Association Implementations	26
3.2.3 Transformation of Associations	40
3.3 Associative Objects	42
3.4 Class/Associative Object Invariants	44
3.5 DDL Generation From AWSOME	46
IV. Generating Data Manipulation Language (DML)	49
4.1 Representation of Expressions in AWSOME	50
4.1.1 WsMethod	50
4.1.2 WsSubprogram	51
4.1.3 WsContainerFormer	51
4.1.4 Pattern Matching	53
4.2 Generating DML From AWSOME	56
4.2.1 Runtime Variables	57
4.2.2 Class and Instance Methods	57
4.2.3 Determining the Return Set	57
4.2.4 Expressing the Set Former Constraint	59
4.2.5 Aggregation	62
4.3 Summary of DML Generation	62

	Page
V. Results, Conclusions and Recommendations	63
5.1 Results	64
5.1.1 Accomplishments	64
5.1.2 Obstacles	65
5.2 Conclusions	66
5.3 Recommendations for Future Work	67
5.4 Summary	69
Appendix A. School Specification in AWSOME Syntax Before DDL Transformation	70
Appendix B. School Specification in AWSOME Syntax After DDL Transformation	74
Appendix C. DDL Generated From School Specification	80
Appendix D. DML Generated From School Specification	83
Bibliography	85
Vita	87

List of Figures

Figure		Page
1.	Transformation Process: Formal Specification-to-Application Code	6
2.	Examples of AWSOME Syntax	8
3.	AST for AWSOME Function <code>getStudentsAdvised()</code>	9
4.	Existence Dependence Diagram	11
5.	School System Example	16
6.	WsClass Specification	17
7.	WsClass Structure	18
8.	WsAttribute Structure	20
9.	WsDataObject Structure	20
10.	Single Inheritance Example	21
11.	Example of a Class After Transformation	23
12.	Example of Inheritance Subclass After Transformation	23
13.	WsAssociation Structure	25
14.	WsAssocEnd Structure	26
15.	WsAssociation Syntax	26
16.	1-to-1 Mandatory Example Design by Migrating Primary Key . . .	28
17.	1-to-1 Mandatory Participation Example Design by Associative Object	29
18.	N-to-1 Both Mandatory Example Design by Migrating Primary Key	32
19.	N-to-1 Both Mandatory Example Design by Associative Object . .	33
20.	Aggregation Implementation Example	35
21.	Associative Object Link Attribute Implementation	36
22.	Qualifier Implementation	38
23.	Ternary Association Implementation	39
24.	M-to-N Association Transformed to Associative Object	41
25.	Association Implemented by Migrating Foreign Key	42

Figure		Page
26.	WsAssocObject Structure	43
27.	Transformed Associative Object Without Qualifier	44
28.	Class Invariant for Multiplicity Constraint of Association	45
29.	DDL Code Automatically Generated From AWSOME	48
30.	AWSOME Specification for Function: getMyFaculty()	50
31.	WsMethod Structure	51
32.	WsSubprogram Structure	52
33.	WsContainerFormer Structure	52
34.	WsLogicalVariable Structure	53
35.	Pattern Matched in a Post Condition	54
36.	WsIn Association Representation Example	55
37.	DML for an Association Implemented as a Foreign Key	60
38.	DML for an Association Implemented as an Associative Object . .	61
39.	DML for Selected Component and Input Variable	61

List of Tables

Table		Page
1.	AWSOME to Relational Type Conversion	47

Abstract

This research creates a methodology and corresponding prototype for the transformation of object-oriented (OO) specifications to represent the corresponding relational schemas that are used to automatically generate database design language (DDL). The transformation design decisions and specifications are then used to generate database manipulation language (DML) that can be embedded within the software application code generated from the same OO specifications. This concept of developing a model for producing compilable and executable code from formal software specifications has long been a goal of software engineers. Previous research at the Air Force Institute of Technology (AFIT) has not focused on the representation of persistent data from OO software specifications. Relational databases are historically among the most popular methods of managing persistent data associated with software systems. However, there is not an automated tool available that will create the DDL and DML from OO specifications. This research develops a framework for combining these separate processes into a single step. Generating the relational database and the operations to manage data within the database from the formal software system specification. When combined with software system code generation, this research will allow the production of entire software systems to include the application code and persistent data management in a relational database.

Generating Executable Persistent Data Storage/Retrieval Code from Object-Oriented Specifications

I. Introduction

The ability to produce executable code from formal specifications has long been a goal of software engineers. This concept is very enticing due to the high costs associated with changes to specifications during software development, resulting in costly code rewrites and extensive time overruns. Additionally, due to the high dollar and human expense of maintaining software after a system is fielded, the ability to maintain a specification and automatically generate the code is very attractive. Another benefit from formally specifying requirements is the ability to mathematically prove the correctness of the specification and consequently the executable code generated through transformations of the specification.

Many organizations and businesses use database systems to store and manage vast amounts of data. Relational databases have been used for this purpose by both the military and business communities for a considerable period of time. As a result, organizations have invested heavily in software and training for relational databases. This, in turn, affects the decisions for future software development. Traditionally, software and database development occurs in separate steps. Software engineers build applications and database administrators are responsible for the database. Links between the two systems must then be built to save and retrieve data. This research developed a framework for combining these separate processes into a single step. Generating the relational database and the operations to manage data within the database from the formal software system specification does this. When combined with software system code generation, this research will allow the production of entire software systems to include the application code and persistent data management in a relational database.

This chapter describes the motivation for automatically generating the structure to support persistent storage of system data and the embedded code to access that data. It

gives information on the background of attempts to generate code using AFITtool. This chapter also assesses past efforts and defines the scope of this research. The final section gives an overview of the rest of this document.

1.1 Background

Research previously accomplished at the Air Force Institute of Technology (AFIT) produced a domain model from formal specifications written in Z (pronounced zed) [19]. This research has included eliciting and harvesting design decisions from the designer based upon information found in the domain model. In addition, work has been done to represent both primitive and aggregate objects found in the domain model and to transform these structures to Ada. This and other research has been incorporated for several years into an ongoing research prototype referred to as AFITtool.

This work has not been extended to the identification, representation, or management of persistent data. Currently, commercial-off-the-shelf (COTS) products such as Rational Rose and ERwin produce Database Design Language (DDL) that, when exported to a Database Management System (DBMS), can be used to create a database within the DBMS [4, 11]. These tools allow users to create schemas for a database system in the form of relations and associated indices via user-friendly interfaces commonly using point and click technology. However, the capability to take a formal specification for a software system and generate the DDL for defining a database and the Data Manipulation Language (DML) for working with the data in a database has not been pursued. In order to reap the benefits of automated code generation, the subject of persistence must be addressed.

1.2 Problem

The goal of this research was to demonstrate the ability to produce a design model based upon the information found in the domain model for persistent data in a software system, and produce DDL and DML from the design model using Knowledge Based Software Engineering (KBSE) methods.

1.3 Assessment of Past Effort

The concept of producing executable source code from formal specifications is not new. At AFIT, the AFITtool System is capable of producing source code from a specification. This is done by incrementally transforming the user's original problem specification into equivalent source code. The subject of persistent data was not addressed in previous efforts. The previous work served as a base for this thesis as the ability to handle and produce code for persistent data in a relational database was added.

1.4 Scope

This research effort was concerned with adding the ability to store and manage persistent data to software systems generated from specification. The target for this research was relational database management systems (RDBMS). Other means of storing persistent data such as file structures and object-oriented database systems (OODBMS) were not addressed. SQL is used to define relational databases and to manage the data they contain. For this effort the target language was SQL92. SQL3, which contains many new capabilities, was still being specified at the time of this research and had not been formally released in a finished form. In addition, the relational portion of SQL92 is the base of the relational portion of SQL3, and the proof of concept provided by this work should be extendable, with minor modifications, to the relational portion of SQL3. The inclusion of new SQL3 capabilities is left to future development of this research.

1.5 Contributions

This thesis implements a solution capable of generating SQL Code from a formal specification using a limited number of expressions. It also deals with the need for persistent data within software systems. It provides a framework for complete generation of SQL, to include triggers and indexes, from formal specification. The implications of this research indicate that through extensions to this methodology, a complete system for generating executable source code with embedded SQL from specifications can be realized.

1.6 Document Structure

Chapter 2, Background, discusses background information on AFITtool and the new AFIT wide spectrum object modeling environment (AWSOME) abstract syntax tree (AST) structure. Started during the same time period as this thesis effort, AWSOME is the next generation of AFITtool and is therefore used throughout this research. Transformation methods between object-oriented (OO) and relational methodologies are also presented. Chapter 3, DDL Generation, examines the additions to the AWSOME structure needed to represent and transform a specification for SQL code generation. In addition, an analysis of associations and primary/foreign key implementations is depicted. Finally, physical DDL generation and the transforms necessary to implement this goal are described. Chapter 4, DML Generation, describes further additions to the AWSOME structure needed to represent the specification of operations. Also included is a description of how specification patterns were matched and combined with DDL generation design decisions to generate DML. Finally, Chapter 5, Results, Conclusions and Recommendations, summarizes the findings of this research and lists areas for future research.

II. Background

2.1 Introduction

This chapter includes background information required to gain a basic understanding of the concepts needed to generate embedded SQL automatically from a formal specification. The first section is background information on AFITtool since it is this continuing effort upon which this thesis is based. The second section discusses two new models used during this research. The third section discusses object-oriented and relational designs and methods of conversion between these two designs. Finally, existing tools for the automated generation of SQL and their shortcomings are described.

2.2 Background Information on AFITtool

AFITtool was created to use formal methods of software development to transform specifications to application code. This tool was built using Software Refinery in the REFIN language. Domain and software specifications are parsed into AFITtool and transformed into functional application code as shown in Figure 1. Currently Z is used to represent the specification formally. Z was chosen because it is a formal declarative language that is very useful for representing domains and specifications [16]. However, any specification language could be used if the proper transformations were added.

A domain or specification written in Z is parsed into an AST called the Domain Object Model (DOM) or domain AST. The DOM is a general representation that can store any type of OO specification. If a domain is parsed into this AST, it is then transformed by the Elicitor Harvester system that helps the designer extract the specification from the domain. This DOM to DOM transformation interactively refines the specification derived from the problem requirements as represented in this AST [1]. The specification contained in the DOM is then converted by a number of design transformations into the design AST or the Generic Object Model (GOM). It is during this set of transformations that state transitions and attribute constraints are changed into methods and the get/set methods are created. Finally, the GOM-to-code transformation takes place [9, 20]. Currently, Ada

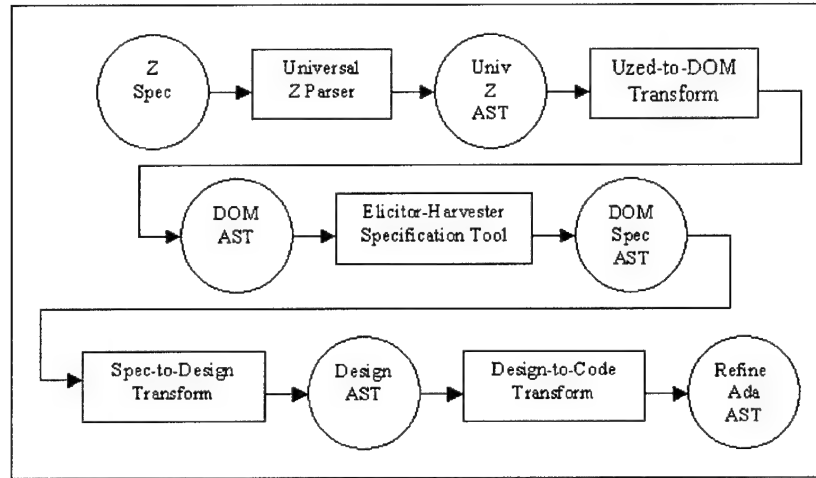


Figure 1. Transformation Process: Formal Specification-to-Application Code

code is being produced and an effort is underway to add Java. However, any language could be produced with modifications to AFITtool's GOM-to-code transformations.

Currently, AFITtool has no way of representing persistent data. This implies that all active object instances will be created during program operation and will cease to exist when the program terminates. To be robust, a specification-to-code system must anticipate the need for persistent data and generate code to organize, store, and retrieve that data efficiently. One manner of performing these actions is with a relational database.

2.3 From GOM Through COIL to AWSOME

The DOM and GOM structure is a product of the integration of previous research efforts. While this structure does work, it requires special logic and operational considerations. A number of DOM to GOM transformations were required since they were not in the same format and the GOM lacked some of the structure of the DOM. Additionally, much straight copying of information from one to the other was required.

During this research, Graham developed a new, simple programming language called the Common Object-Oriented Language (COIL) at AFIT [7]. This served a number of purposes. Its primary focus was to act as a single common language for imperative programs written in other languages. It was also for reverse engineering and language trans-

lation. Another goal of this language was simplicity in that most commercial languages are extremely complex and idiosyncratic. Since all imperative languages have certain core features in common, this language captured those common aspects in an uncomplicated manner. This language was small and simple, yet complete, and considered an excellent alternative to previous research efforts. However, this language still required transformations to the GOM for executable code generation as it was intended to replace the GOM.

Another member of the KBSE research group proposed combining the COIL with the functionality of the GOM. This new model, AWSOME, used the COIL as the base language. COIL was then expanded to have the capability to capture the information stored in both models. Under this new model, transformations take place within the same representation thus deleting the need to create special syntactical transformations between models. Additionally, AWSOME provides a robust environment for software synthesis and reverse engineering while still providing a complete language [3].

AWSOME is capable of handling packages, declared classes, variables, associations, methods, and other common programming constructs. The language model is intuitive as it is broken down into packages that contain a series of declarations. These declarations include data types such as classes, objects including variables and constants, and subprograms that contain procedures and functions. Examples of the AWSOME syntax are shown in Figure 2. While the syntax is simple and intuitive, the corresponding tree structure, as captured in the AWSOME AST, is not as intuitive. An example of a simple function is shown in Figure 3. The language supports object-oriented concepts such as inheritance, aggregation and polymorphism. It does not support input/output as most application languages handle this in ways that cannot be specified generically to meet the language-specific intricacies that are often contained in very large complex libraries. This research effort is done within the AWSOME environment.

As is depicted in Chapters III and IV, a significant portion of this research involved the development of portions of the AWSOME AST and the development of transformations that act upon it. This single tree representation allows not only transforms, but also the capture of design decisions behind them that are necessary for the handling of persistent data in a relational database.

```

type Integer      is range -100000 to 100000;
type Date        is array [1..9] of Char;
type StudentSet  is Set of Student;

Class Person is
  var lastname      : String;
  var firstname     : String;
  var initial       : String;
  var birthdate     : Date;
  var ssn           : String;
  var height        : Integer;
  var weight        : Integer;
end Class;

Class Faculty is Person with
  var academicRank  : String;

  function getStudentsAdvised() : StudentSet is
    guarantees getStudentAdvised =
      {s | ( s : Student) s in
this.RAdvices.Advisee };
end Class;

```

Figure 2. Examples of AWSOME Syntax

2.4 Transformation Methods Between Object-Oriented and Relational Methodologies

Relational databases are commonly used as a means to store data throughout the business world and the Air Force. This has been one of the primary methods to store data that is to be queried and updated on a regular basis. The object-oriented methodology is a primary technique taking the software development community by storm. Object-oriented programming attempts to model software problems in terms of real world objects. Its benefits include software reuse, inheritance, and encapsulation [13]. However, as new applications are built, rather than using an OODBMS designed specifically for OO languages, these applications often must use an RDBMS either because it is the tool of choice, or because of existing software and database investments. Traditionally, this type of software development and its coupling with an RDBMS has proceeded along the following lines.

1. Design the application.
2. Devise a specific architecture for coupling the application to the RDBMS
3. Select a specific RDBMS (in reality, this step often occurs first).
4. Design the database with RDBMS code to set up the proper database structures.

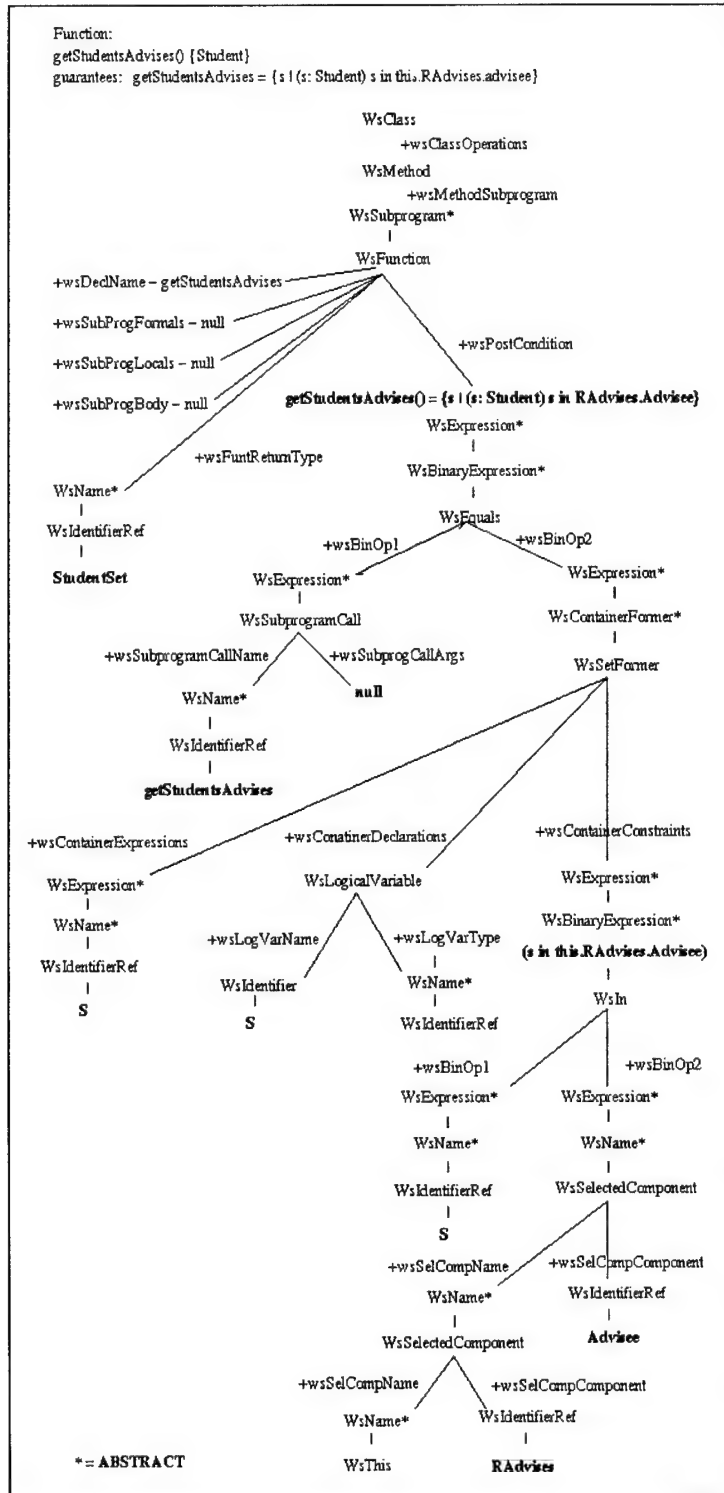


Figure 3. AST for AWSOME Function getStudentsAdvises()

5. Write programming language code to compensate for RDBMS shortcomings; provide a user interface; validate data; and perform computations.
6. Populate the RDBMS with data.
7. Run the software application. Query and update the RDBMS as needed [17].

A number of different automated or semi-automated formal methods have been proposed to translate RDBMS to OODBMS [6,8]. However, there is not the same plethora of methods for formal translation of OO schema to relational schema as required for OO programming language development with an RDBMS. This is because the OO schema includes information that is not contained within the relational model, such as inheritance, aggregation, and the inclusion of methods [5].

The cardinality of the relations has been shown to be equal to object-oriented associations' multiplicity constraints. These constraints can be mapped to four properties: functional, injective, surjective, and total. Relationships whose cardinality does not map directly, such as 1-to-2, can be derived from these four properties [2]. Additionally, referential integrity deficiencies may be supplemented in RDBMS by using triggers. Triggers may also be used to enforce complex business rules and to audit changes to data. They are executed by the database when specific types of data manipulation commands are performed on specific tables. These are transparent to the user and include commands such as inserts, updates, and deletes. Triggers work individually or in groups. However, triggers should be used only to supplement referential integrity and not as a substitute for it [10].

There are numerous informal case tools for designing the relational database as listed in Step 4. There are also tools for developing the embedded SQL necessary for the software and the RDBMS to function as mentioned in Step 6. In addition, there is an excellent tool for finding mathematically equivalent SQL statements and replacing them with more optimal functioning statements.

2.4.1 Informal Case Tools for Automated Development of DDL. ERwin by Logic Works is a traditional entity-relationship design tool. This tool is representative of a whole family of similar products. It combines a windows graphical interface and editors to define database objects. ERwin helps design RDBMS using the standard IDEF1X diagramming method developed by the US Air Force or an information engineering notation [11]. The

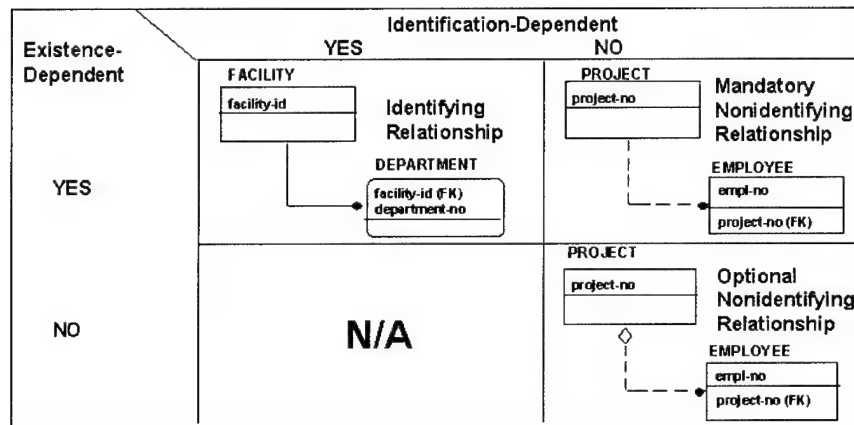


Figure 4. Existence Dependence Diagram

objective of this tool is to create an easy to understand graphic representation of the relational schema. Once this is created, ERwin builds the physical database schema by generating a DDL script for the selected commercial database. This tool requires the designer to identify all objects and all corresponding key as well as non-key attributes. Then the relationships between the objects are defined in terms of Existence Dependent and Identification Dependent as shown in Figure 4. The cardinality of the relationship must also be defined. Once all this is done, ERwin automatically generates the DDL to create the physical database.

Rational Rose is another design tool. This tool is broader in scope as its purpose is OO software systems development. It allows development in Ada 95, Java, C++, Corba/IDL, and database schema generation in DDL. Rose supports the Unified Modeling Language (UML), Booch, and Rumbaugh notations, making it more flexible [4].

This tool requires the designer to identify all objects and the corresponding attributes. Each attribute must then be defined in terms of type, length, nullability, primary key, unique, composite unique, and constraints. Rose treats associations as bi-directional relationships that denote a semantic dependency between two classes. If an association exists between two persistent classes, a relationship will exist between the resultant generated tables. Multiplicity of the association is commonly used as the determining factor for the creation of relations between classes. However, if he chooses, the designer can determine the relation via the direction of the association regardless of multiplicity. In the case of

inheritance, the superclass always migrates its primary key to the subclass. For aggregation or containment, Rose adds a DELETE CASCADE constraint to the foreign key in the database. For M-to-N associations, a link class/associative object, is used. However, the user must create the link class as Rose will not make this logical conclusion on its own [15].

While this tool does allow the designer to create an informal OO schema and generate a corresponding relational database schema, it makes design assumptions that might not be ones desired. Since the tool uses only the multiplicity of the association to determine how to design the relation in the database, it is limiting and may not capture the true nature of the relation. In addition, this tool produces only the DDL portion of SQL.

2.4.2 Informal Case Tools for Automated Development of DML. There are also numerous informal case tools that can be used to help semi-automatically generate DML. These tools are designed to generate, test, and analyze embedded SQL. They have similar operational requirements and attempt to make the creation of SQL simple for the designer by using a series of pull-down menus and selection boxes. Using this interface, the designer must select the tables on which the operation is to be performed. Next, the type of operation is selected: SELECT, UPDATE, DELETE, or INSERT. Conditions upon the fields such as equals, greater than, like, etc. can then be added. In addition, data organization commands such as group-by and order-by may be placed within the operation. Once a given statement is completed, the tool then creates an embedded DML statement in the language of choice and returns a sample result set. Often these tools will provide extra data to the designer such as the order of operations, joins, unions, or specific indexes used by the optimizer to get the result set. In this manner, the designer may visually determine whether the statement is acting as desired. The designer can then determine whether the DML statement meets the original software system specification. There is no mathematical determination whether the statement is meeting a formal specification [14, 18].

2.4.3 Formal DML Transformation Tool. LECCO SQL Expert by LECCO Technology Limited is a tool for automating SQL transformations. This product attempts to automatically optimize SQL with transformation capabilities. It scans either application files (text or binary) or database objects searching for SQL statements. Found statements

are then categorized as simple, complex, or offensive in that they are not valid statements. Another LECCO capability is optimization of SQL statements. This option produces a list of semantically equivalent SQL statements, through formal transformations, that provide better performance. This tool uses a parser and artificial intelligence to guide the SQL generation. When a statement is selected for optimization, the tool will indicate the number of equivalent statements it has found and tested that produced different optimization plans. The only alternate statements which are kept are those that result in a unique optimization plan. For one complex statement tested, the system investigated 180 equivalent statements but only returned 80 of them. The other 100 were eliminated because their optimization plans were already created. Also, all statements that took longer to run than the original were eliminated without ever being added to the list of 180 equivalent statements. A ranking was then produced according to oracle cost. This may not be the best-cost option when operating on the actual software system. For instance, if bind variables are used, the actual operating time may vary greatly. The only true way to test the statements is to run them on the actual system to see which option executes the fastest. This system transforms mathematically equivalent SQL statements once an original statement is entered. However, this system is not capable of determining whether the original statement was specified correctly. It can only operate on what is assumed to be a correctly specified SQL statement [12].

2.4.4 Conclusion. The tools described fall short of the goal of this research to automatically add database capability to software systems generated from specification. The tools such as ERwin and Rose help the designer develop the database schema. Other tools provide simple user interfaces for developing database management operations. However, in both cases, these tools look only at a small portion of the problem. The tools that help write DML are particularly lacking as they only provide a user interface for developing SQL. They do nothing to ensure the operations created meet specified requirements. It is left completely to the designer to visually determine that the code generated performs the operation specified. Finally, the formal tool by LECCO Technology can only create equivalent statements of those already in place.

This research combines the capability of the DDL generation tools with the generation of DML from the specification. By combining these capabilities, the DML operations can be guaranteed to meet the specified requirements. In addition, these previously separate operations are combined into a single automated process. This allows the generation of complete software systems to include the application code and persistent data management in a relational database.

III. DDL Generation

The design of a relational database includes two main steps: defining the structure and manipulating it to implement specified requirements or functions. The first is done in DDL while the second is done in DML. This chapter covers DDL. In this research, the input is an OO specification written in AWSOME. In order to generate DDL, this specification must be transformed into its relational equivalent. As stated previously, DDL generation from an informal specification has been performed and is generally available from numerous COTS producers. However, since the design decisions from developing DDL are needed to generate DML, DDL generation is a necessary part of this research. The DDL generation is not meant to be competitive with COTS products. It is meant to produce executable DDL code and document design decisions. This research generates DDL from formal specifications while the COTS products do so from informal specifications.

The generation of DDL is performed in two phases. The first is to transform the specification within the AWSOME AST. The second is to read the specification and design decisions to generate a string that when applied to a database will create the correct table structure. The first phase of transforming a specification from object-oriented to relational schema is done by performing three groups of transforms on the specification as represented in AWSOME. These transformations can be categorized as follows: the classes are mapped to tables; the associative objects are mapped to tables; and the associations are designed into database relations. For all of these transformations, additional information is needed from the designer. These transformations are based on the assumption that the specification is correct and properly parsed into the AWSOME tree structure.

The transformation of classes, associative objects, and associations can be an iterative process depending upon the specification and the design implementation. For instance, in the school system example, Figure 5, a number of classes, an associative object, and an association must be transformed in order to transform the class `Section`. This class is then used to migrate a key to the associative object, `AssignedAO`, as part of an association implementation. The foreign key then becomes part of the primary key of the receiving associative object. This small example shows the complex procedure of designing a specification to have persistent storage in a relational database. Once these design decisions

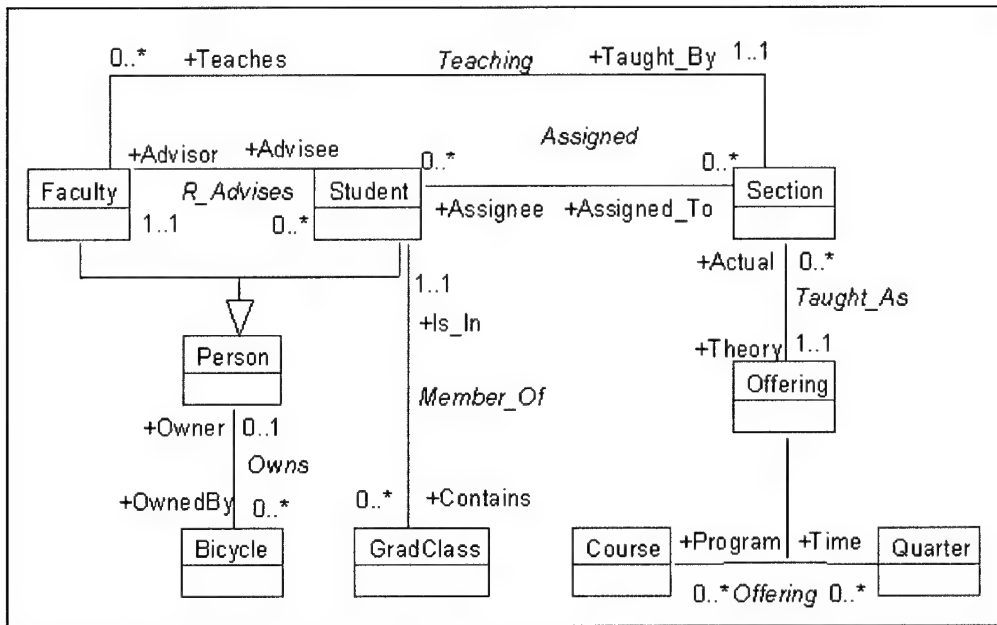


Figure 5. School System Example

are made and recorded for future DML generation, DDL can be generated from the AWSOME structure. These transformations are destructive in that if reversed, they cannot be guaranteed to return to the original specification, but only to a mathematically equivalent specification.

After these transforms are performed, the specification as held in the AWSOME structure is ready for the second phase of DDL generation. The specification is not transformed further during this phase of DDL generation. Instead, the tree is traversed and DDL is generated from the existing specification. The relations between all the class/associative objects are examined to ensure there are no circular dependencies. If none are present, each class/associative object is mapped to a table. Each object's attributes are mapped to columns within its respective table. The datatype of each attribute is converted and assigned to a datatype that the DBMS recognizes. Next, each attribute is checked to determine whether or not it is part of either the primary key or a foreign key. If so, statements are added to the table definition to indicate this information. Once a definition of a table is created as a string, it may be applied to the database. In this manner DDL is generated.

The AWSOME syntax for specifying the classes, associations, and associative objects as they relate to DDL generation is described in this chapter. The structural components necessary to capture the specification and the design transformations are examined. Object-oriented associations differ from the relations of RDBMSs. Since this difference can cause many problems, an analysis of association implementation within an RDBMS is performed. Finally, the implementation of DDL code generation is depicted.

3.1 Mapping Classes to Tables

The specification for a class, Figure 6, is contained within the AWSOME structure `WsClass` as depicted in Figure 7. For this research, three items from this structure are of interest: the attribute `wsAbstract` indicates whether the class is persistent or abstract; the component `wsClassDataComponents` contains the attributes of the object; and the component `wsClassSuperclass` names the superclass of the object.

3.1.1 wsAbstract and Class Persistence. A table is created for each persistent class. In this research, all classes were treated as persistent. However, in general, at the class/object level, there must be a means to determine whether the class is persistent. In the simplest case, tables are created to hold the instances of persistent classes. Otherwise, no table is required, and the instantiation of these objects occurs only during execution of the program. In case of a system crash, this data may not be recoverable. The boolean flag, `wsAbstract`, was created within the `WsClass` structure to indicate whether an object

```

Class Person is
  var lastname      : String;
  var firstname     : String;
  var initial       : String;
  var birthdate     : String;
  var ssn           : String;
  var height        : Integer;
  var weight        : Integer;

  function getMyBikes() : BikeSet is
    guarantees getMyBikes =
      { b | { b : Bicycle) b in this.Owns.OwnedBy };
  end Class;

```

Figure 6. WsClass Specification

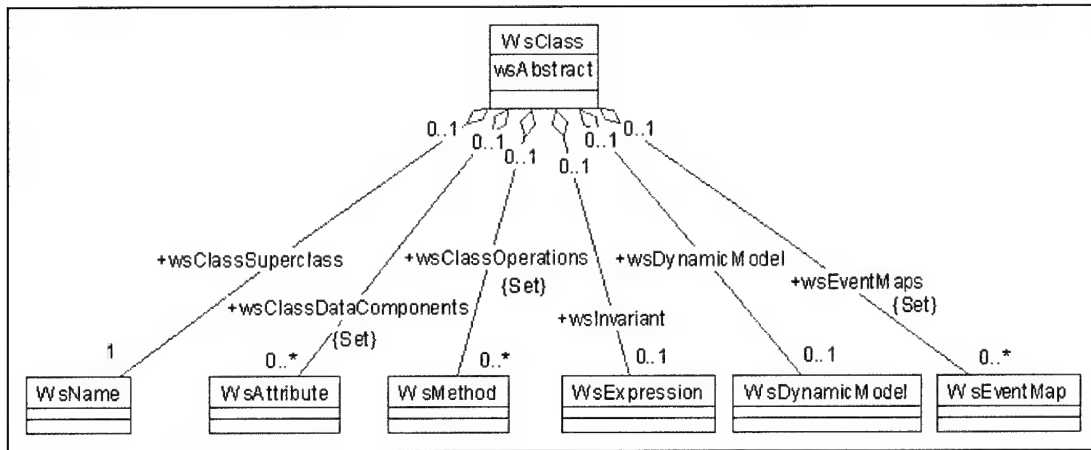


Figure 7. WsClass Structure

is persistent or abstract. The default value for this flag is false, indicating the class is concrete. If the flag is true, the class is considered abstract, and a corresponding table in the database is not created. There are situations however, that cannot be handled by this attribute alone. If only some instances of a class are to be saved in the database while others are not persistent, the software application needs to specify those instances and include explicit data control operations such as database create commands or explicit save options. This effort handles only the cases where all instances of a class are to be persistent.

Additionally, a question arises at the class/object level. Does an abstract class with attributes, that is not instantiated, need to be represented by a table within the database? If the abstract class is part of an inheritance chain that leads to a concrete class, then it should be represented by a table with a column for each attribute defined in that class. Inheritance as a whole and its representation within the relational database are discussed in this chapter. However, in this specific case, inheritance is represented in RDBMSs by the first method described in Section 3.1.3. A relational JOIN of all of the tables in a path to the root with the common key as the join parameter retrieves an instance of a concrete subclass. By doing this, the inheritance structure is maintained while handling abstract classes as a separate entity without blending their attributes into concrete classes. This case should be handled by declaring the abstract superclass as concrete for RDBMS

purposes in the specification. The question of persistence for classes is handled in this manner.

3.1.2 Attributes. Within the `WsClass` structure, the component `wsClassData-Components` is implemented as a vector of `WsAttributes`. The class `WsAttribute`, Figure 8, is needed to capture the attributes of an object. Three of the four boolean flags in this class are for DDL generation. `wsUnique` identifies whether the values within the attribute's column in the object's table must each be unique. Any field with this identifier is a candidate to be an alternate key. A simple primary key would also have this constraint. However, if a field is part of a complex primary key, it may not have this restriction. The complex primary key in this case would have an implied combined uniqueness constraint. Another constraint that may be applied in conjunction with uniqueness, but may also occur alone, is the possibility that a field can hold null for a value. Declaring the field nullable with `wsNullable` fulfills this requirement. The final identifier needed to mark whether a field is part of the primary key is the boolean element, `wsPrimaryKey`. This indicates that the attribute is the primary key, or part of it, for the table representing the data object being transformed.

The `WsAttribute` also contains two elements that are not boolean. `wsAttribute-HomeClass`, a `WsIdentifierRef`, is a pointer to the class from which the attribute originally came. This is a field specifically added for foreign keys. When a foreign key is added to a class, this field indicates from which class the transfer originated. `wsAttributeDataObject`, a `WsDataObject`, contains the name, datatype, and initial value of the attribute.

`WsDataObject` extends `WsDeclaration` and represents a data object. It is abstract and is instantiated either as variable by `WsVariable` or as a constant by `WsConstant`. `WsDataObject` inherits the element, `wsDeclName`, which is a `WsIdentifier` that holds the data object's name. This structure also has two elements of its own as shown in Figure 9. The element, `wsDataObjectValue`, is a `WsExpression`. This element is the value of a constant or the initial value of a variable, depending on the instantiation. `wsDataObjectType` is a `WsName` that references the data type of the attribute. Within the `AWSOME` structure, it is possible to define subtypes with constrained ranges. Some RDBMSs allow a field to

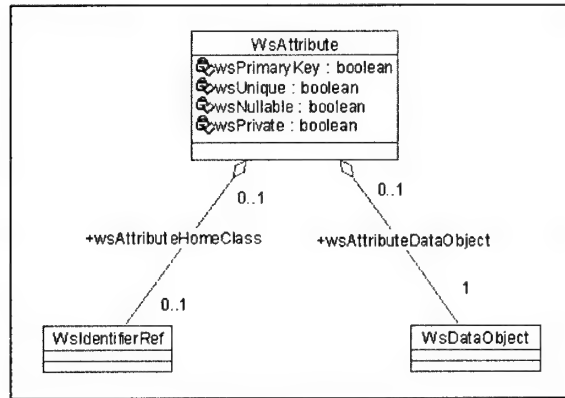


Figure 8. WsAttribute Structure

be constrained to a certain range of values when the table is created. An example is that the birth date on a driver's license must be at least 16 years prior to the entry date. If this capability exists within the RDBMS, AWSOME attribute type constraints could be transformed into column constraints. However, since syntax and the ability to use column constraints are not global throughout all RDBMSs, this functionality is not implemented in this effort.

3.1.3 wsClassSuperclass and Specifying Inheritance. Inheritance is often considered a meta-association in the object-oriented paradigm. If inheritance is present, the element, *wsClassSuperclass*, a *WsName*, references the object's superclass. This *WsName* refers only to a single object since AWSOME only supports single inheritance.

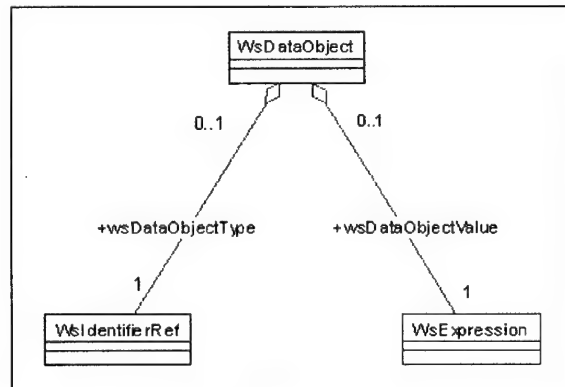


Figure 9. WsDataObject Structure

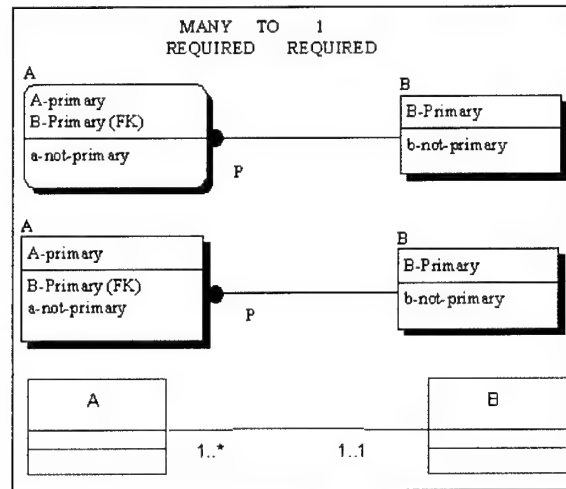


Figure 10. Single Inheritance Example

There are more ways than one to transform inheritance. To give a full understanding of the implications of the different methods, an analysis of these methods is performed.

The preferred method of handling single inheritance, as depicted in Figure 10, is to migrate a key. In this case, each superclass and subclass maps to its own table, and a foreign key from the superclass is added to the subclass. This method is the most common, follows the normal forms, and logically follows the inheritance class structure. Another advantage is that the number of different subclass objects can easily be changed without affecting the rest of the database schema.

Another method is to move all the superclass's attributes to the subclass's table. This method still requires a table for each subclass. The superclass table is absorbed into each subclass table. This method could be used if the superclass has few attributes and the subclass has many attributes. This approach may still enforce third normal form. However, a problem with this method is the loss of the superclass/subclass structure. The database schema loses this level of abstraction. Additionally, each table may need to be searched separately when trying to locate an instance of an object. For example, when the ID number of a piece of equipment is known, but not the nomenclature, which table to search is unknown. Use of this method would depend strictly on the designer and the application.

In the third method, the superclass table has all the subclass's attributes added into its table. This method can be used if the superclass has many attributes and the subclass has few attributes. This approach does not enforce third normal form. In addition, space is wasted for the different subclasses because each inserts different fields. This method should be used only when there are very few subclasses. The structure reflects differences in types of objects by the fact that separate fields are null in different instances of the object. As more subtypes are added, this method wastes more space due to null fields for non-applicable instances. Of these choices, the first method is the only one employed within this research for the reasons stated above.

3.1.4 Transformation of Classes. The instances of `WsClass` are transformed to ensure the identification of a primary key and to implement inheritance if it exists. For this effort, all classes are assumed to be persistent regardless of whether or not they are abstract. Consequently, all classes have a table representation created during the DDL generation phase. All defined classes are editable in that any attribute may be identified as a primary key. If more than one attribute is identified, then the key is considered a composite primary key composed of numerous fields. Any attribute identified may be a native attribute or one imported as a foreign key from a different class as shown in Figure 11.

To transform each class, the primary key, simple or complex, must be identified from the attributes of the class. To identify a primary key, the boolean attribute, `wsPrimaryKey` of `WsAttribute`, is set to true. The designer accomplishes this by selecting the class to be transformed and then individually selecting each attribute that is to be identified as the primary key or part thereof. When the class being transformed has a superclass, the inheritance is implemented by migrating the attribute identified as the primary key in the superclass to the subclass. This foreign key will act as part of, or as the entire primary key of the table generated from the subclass. An example of an inheritance subclass after transformation, showing the key migration, is presented in Figure 12.

Additionally, if the class is involved in an identifying mandatory association that the designer desires to implement by migrating a key, the class receiving the key should use

Before Transformation		
The Attributes of the Class: Section		
<u>Primary Key</u>	<u>Name</u>	<u>HomeClass</u>
0. false	snumber	null
1. false	TaughtAs__Offering_Course_cnum	Offering
2. false	TaughtAs__Offering_Quarter_qname	Offering
3. false	Teaching__Faculty_Person_ssn	Faculty
After Transformation		
The Attributes of the Class: Section		
<u>Primary Key</u>	<u>Name</u>	<u>HomeClass</u>
0. false	snumber	null
1. true	TaughtAs__Offering_Course_cnum	Offering
2. true	TaughtAs__Offering_Quarter_qname	Offering
3. false	Teaching__Faculty_Person_ssn	Faculty

Figure 11. Example of a Class After Transformation

Before Inheritance Transformation		
The Attributes of the Class: Faculty		
<u>Primary Key</u>	<u>Name</u>	<u>HomeClass</u>
0. false	academicRank	null
After Inheritance Transformation		
The Attributes of the Class: Faculty		
<u>Primary Key</u>	<u>Name</u>	<u>HomeClass</u>
0. false	academicRank	null
1. true	Person_ssn	Person

Figure 12. Example of Inheritance Subclass After Transformation

that key as part of, or as the entire primary key of that class. In this effort, mandatory association is not checked. It is up to the designer to ensure that the association is not redefined by assigning a key from a non-identifying or non-mandatory association as part of or as the entire primary key. Furthermore, the mandatory or optional nature of the association is not implemented through the use of the nullable field. Currently, the NO NULL option is set by use of `wsNullable` only for parts of the primary key.

By performing these transformations, the designer has defined a primary key for each table. In addition, migrating a mandatory key to the subclass automatically enforces inheritance. The designer may need to perform other transforms first. For instance, if a class's primary key comes from an association in which it is involved, the association must be transformed first and then the primary key set. Other than in the case of inheritance, it is totally up to the designer to select the foreign keys in a class.

3.2 Associations

Previous research by Kissack at AFIT modeled associations as binary Z relations. These associations were specified over components of an aggregate class. The implementation of associations was limited to only associative objects [9]. In contrast, this research models associations as stand alone entities. Their degree may be binary, ternary, or higher order. Furthermore, these associations may be implemented by numerous methods.

3.2.1 Representation of Associations in AWSOME. The structure, `WsAssociation`, as depicted in Figure 13, is used to represent associations. This structure extends `WsDeclaration` and is placed in the hierarchy tree at an equivalent level to `WsClass`. This is a change from previous efforts. In `AFITtool`, a composite class containing complex attributes of other classes declares an association between its complex attributes. In this research, an association is not a component of any given class or datatype, but rather shows a relationship between classes. By declaring an association as a separate entity, it can be handled as a structural component much like a class. Under this model, aggregation then becomes just another association. This implementation will be discussed later under Association End.

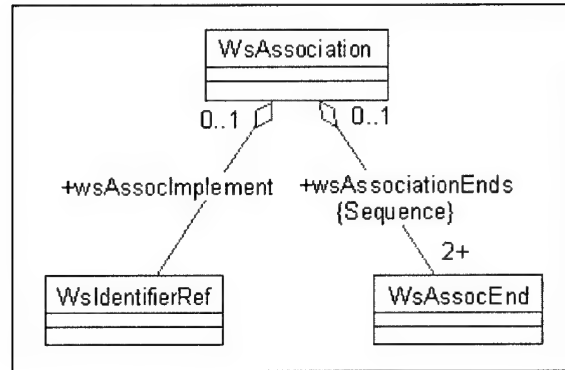


Figure 13. WsAssociation Structure

wsAssociationEnds is a sequence of **WsAssocEnd**. This sequence must have at least two members and represents the ends of an association. This allows for ternary or higher order associations to be represented. **wsAssocImplement** is a **WsIdentifierRef**. This pointer is set, during the design transformations, to the object that implements the association.

WsAssocEnd represents the connections of an association. It has two boolean components and four other elements. This is shown in Figure 14. **WsOrder** is a boolean that indicates whether the end is ordered. **WsAggregate** indicates whether the association is an aggregate relation. In this situation, setting the value to true indicates the Association End that contains the aggregate. By using this flag, it can be determined whether an aggregate association exists and which class has the aggregate component. **wsAssocEndRole** is a **WsIdentifier** and is the role name of the association end. **wsAssocEndClass** is a **WsIdentifierRef** and points to the class to which the association end is connected. **wsAssocEndQualifier** is an optional **WsIdentifierRef** that points to an attribute of an associative object that is acting as a qualifier. This will only be present in associative objects and is explained in Subsection 3.2.2.6. Associations cannot have qualifiers in this model. **wsAssocEndMultiplicity** is a set of **WsIntegerType** that is implemented as a vector. Each **WsIntegerType** has an upper and lower bound. In this manner, the cardinality of the association end can be determined. For instance, it could be 0-1 and 7-9. This structure allows for the inclusion of ranges that are not continuous.

The AWSOME syntax for representing associations is intuitive and simple to read as shown in Figure 15. The association is specified as a stand-alone entity. The name of

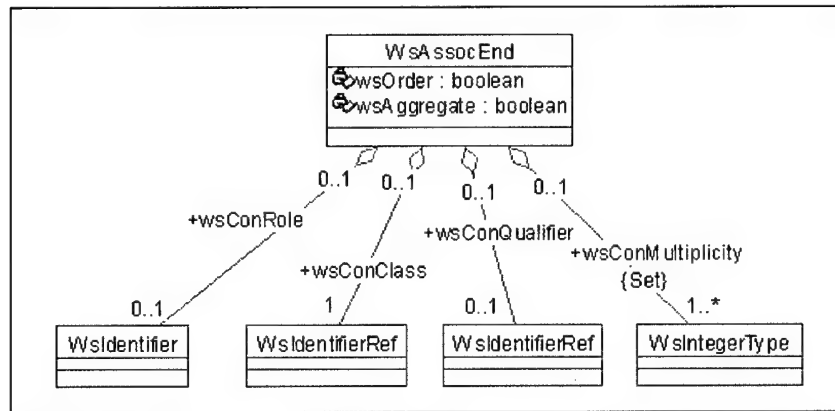


Figure 14. WsAssocEnd Structure

the association is not associated with any other object. In the declaration of the roles, each end/role is self-contained in that all information including role name, object type, cardinality, and aggregation are all present in a single statement. Each association is required to have a unique identifier. However, the roles are only required to have unique names within any given association. Any identifier may be used for a role name, but custom dictates that the name makes logical sense within the context of the association.

3.2.2 Analysis of Association Implementations. Previous efforts at AFIT handled all associations by making them into associative objects. In a relational schema, there are other methods that may be employed such as migrating keys and combining tables. This section examines the associations recognized by this effort and explains the options for handling them.

3.2.2.1 1-to-1 Associations. Previous research recognized three different types of binary 1-to-1 associations [9]. These are differentiated by the mandatory/optional participation of members of the association. In these associations, direction was implied

```

Association RAdvised is
  role Advisor      : Faculty      is oneTo1;
  role Advisee      : Student      is zeroToN;
end Association;
  
```

Figure 15. WsAssociation Syntax

by the order in which the members of the association were expressed, left to right, domain to range. These associations include:

Partial Injection	- Both Optional
Total Injection	- One Optional, One Mandatory
Bijection	- Both Mandatory

Current research no longer uses this representation. Rather than having an association direction with a defined domain and range, the association is defined by its multiplicity and purpose. Additionally, the ends use role names for class/associative object identification and association traversal.

The primary method for representing 1-to-1 associations between objects that will be transformed into a relational database is to migrate the primary key from one object into the other as a foreign key. Which object receives the migrated key can be recommended by an automated system, but must be determined by the designer. This method works well for queries on the association. However, the design decision of which object gets the foreign key must be retained. Otherwise, there will be problems if the relationship is queried in the wrong table. Depending on the association, this method may fail to reflect the mandatory/optional participation in the association of objects without the use of database triggers. Additionally, the 1-to-1 association cannot be determined from the resulting database structure alone. However, this method intuitively shows the association between objects.

In an association that is optional for both objects, the primary key from either object can be migrated into in the other object as a non-identifying key. This means the migrated key is not part of the other object's primary key. Also, the migrated key must be nullable so that the object in which it is present can exist alone without the association.

An association that has mandatory participation for only one of the objects can be expressed in one of three manners. If that object is identification dependent, the primary key from the other object makes up part of the complex primary key for the dependent object. The second choice is a non-identifying association. In this case, the mandatory object has a unique key that does not require the addition of the other object's primary key. The primary key is placed as a normal, non-nullable attribute within the receiving

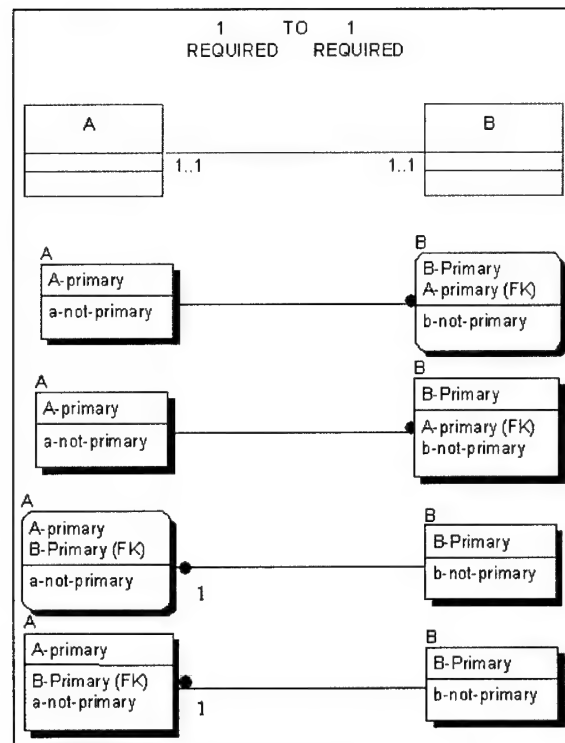


Figure 16. 1-to-1 Mandatory Example Design by Migrating Primary Key

object. The third option is to have the mandatory object's primary key placed as a non-identifying foreign key in the other object's table. This option, however, does not reflect the necessary participation in the association by the mandatory object.

In an association where both objects have mandatory participation, the association can be expressed in one of four ways as shown in Figure 16. If one object is identification dependent, then the primary key from the other object makes up part of the complex primary key for the identification dependent object. The second choice is non-identifying association. In this case, one object has an identifying key that does not require the addition of the other object's primary key to be unique. The primary key is placed as a normal, non-nullable field within the receiving object's schema. The third and fourth options are the same as previously described with the objects exchanging roles.

The next best method to represent 1-to-1 associations, and method of choice in certain circumstances, is to use an associative entity, Figure 17. An associative entity is created by taking the primary keys from both of the objects in the association to form a

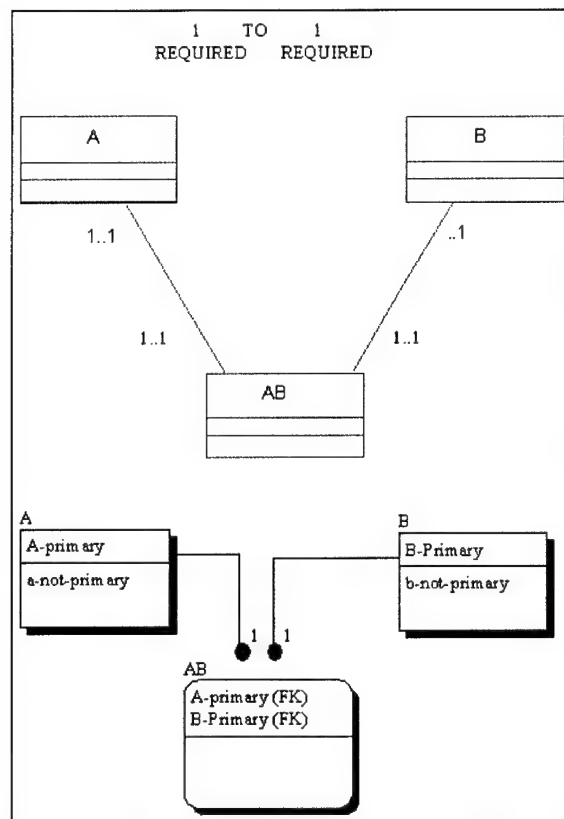


Figure 17. 1-to-1 Mandatory Participation Example Design by Associative Object

complex primary key for the new object. Generally, this is the primary method used when the association has attributes or when the association has a qualifier. These situations will be covered in depth in their specific subsections. Each instance of the associative object represents one instance of the association between the primary objects. This table promotes easy association retrieval by searching the associative object and then using the results to search the primary tables. This method requires more overhead than the first method, as a third table must be maintained. Consequently, all association queries that require more data than the primary keys require at least two table lookups. Furthermore, very little information concerning the type of relationship represented by the associative object can be deduced from the table structure without the use of triggers.

The last choice is to use a single table to represent both primary tables and the association. This method requires the least code to create. However, it lacks association

structure and the application software must handle all management. Also, this method is a poor choice because it results in schemas which violate lower normal forms, and if data entry errors are not detected, cycles in the data dependencies could be created. However, there is one case in which this method might be used. This case is the 1-to-1 mandatory association since there is no wasted space in the table. Either primary key could be used or both together. If this method is chosen, then the association might need to be examined to ensure that there are actually two objects with a mandatory 1-to-1 association and not just a single object. This option is not used in this research.

3.2.2.2 N-to-1 Associations. In previous research, N-to-1 associations depended, much like 1-to-1 associations, upon the direction of the association and the optional/mandatory participation of the objects [9]. Four different binary association types were recognized as N-to-1 associations. These are differentiated by the mandatory/optional participation of members of the association. These associations include:

Partial Function	- Both Optional
Total Function	- N Mandatory, One Optional
Partial Surjection	- N Optional, One Mandatory
Total Surjection	- Both Mandatory

Current research no longer uses this representation. Rather than having an association direction with a defined domain and range, the association is defined by its multiplicity, cardinality, and purpose. The ends use role names for identification and association traversal. In addition, the Total Function and Partial Surjection are considered the same case in this research.

The primary method for representing these four associations between objects that will be transformed into a relational database is to migrate the primary key from the 1 cardinality to the N cardinality object as a foreign key. The designer must determine whether the migrated key is placed into the N cardinality's primary key, thus forming a complex primary key, or in the attribute section. This can be recommended by an automated system depending upon the particular association. This method works well in queries asking for the association from the 1-to-N as well as from the N-to-1. If a query is processed for an object that is not a member of the association, an empty set will be

returned. This method reflects the mandatory/optional nature of the association from its structure in the database. However, the 1-to-N representation cannot be determined from the structure alone.

In the association where participation is optional for both objects, the key from the single object's table must be placed as a non-identifying attribute, not part of the primary key, in the multiple cardinality object's represented schema. Additionally, the migrated key must be nullable so instances of the object can exist without being in the association. Since membership is optional for both objects, the association is modeled as a 0-1 to 0-N entity relationship.

In an association where the N object has mandatory participation, the association can be expressed in one of two ways. If the N object were identification dependent, then the primary key from the 1 object would make up part of the complex primary key for the N object. The other option is a non-identifying association. In this case, the N object has an identifying key that does not require the addition of the 1 object's primary key to be unique. The 1 object's primary key is placed as a normal, non-nullable field within the N object's table. In the inverse case where the association is mandatory for the 1 cardinality object and optional for the N cardinality object, the primary key from the 1 object is placed as a non-identifying nullable foreign key in the N object's table.

For the mandatory participation of both objects, the association can be expressed in one of two manners as shown in Figure 18. If the N object is identification dependent, then the primary key from the 1 object makes up part of the complex primary key for the N object. The other alternative is non-identifying association. In this case, the N object has an identifying key that does not require the addition of the 1 object's primary key to be unique. The 1 object's primary key is placed as a normal, non-nullable field within the N object's table.

The second choice of method for representing N-to-1 associations is Associative Object. This is depicted for an N-to-1, both mandatory association in Figure 19. This is the method of choice when the association has attributes or when the association has a qualifier. This will be discussed further in Subsections 3.2.2.5 and 3.2.2.6. This method

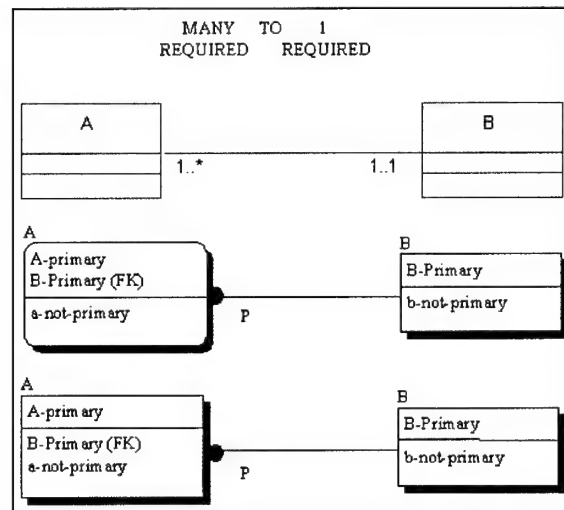


Figure 18. N-to-1 Both Mandatory Example Design by Migrating Primary Key

can also be used when the association is optional for both objects. Each instance of the associative object represents one instance of the association between the primary objects. This table promotes easy association retrieval by searching the associative object and then using the results to search the primary tables. An associative object requires more overhead than migrating a primary key. Consequently, all association queries requiring more data than the primary keys must include at least two table lookups. Furthermore, very little information concerning the type of relationship represented by the associative object can be deduced from the table's structure. The primary key from the N cardinality object will occur at most one time in the associative table. Meanwhile, the primary key from the 1 cardinality object may occur none, one, or many times in the table, depending upon the type of association.

The last choice for N-to-1 associations is the use of a single table to represent both primary tables and the association. This method requires the least code. It is best used when the association is very dense because both the object instances are held in a single tuple. When the association is sparse, much space is wasted because only one object's attributes are filled in any given record. However, for every instance of the association, the 1 cardinality object's fields are duplicated, still causing a waste of space. Further-

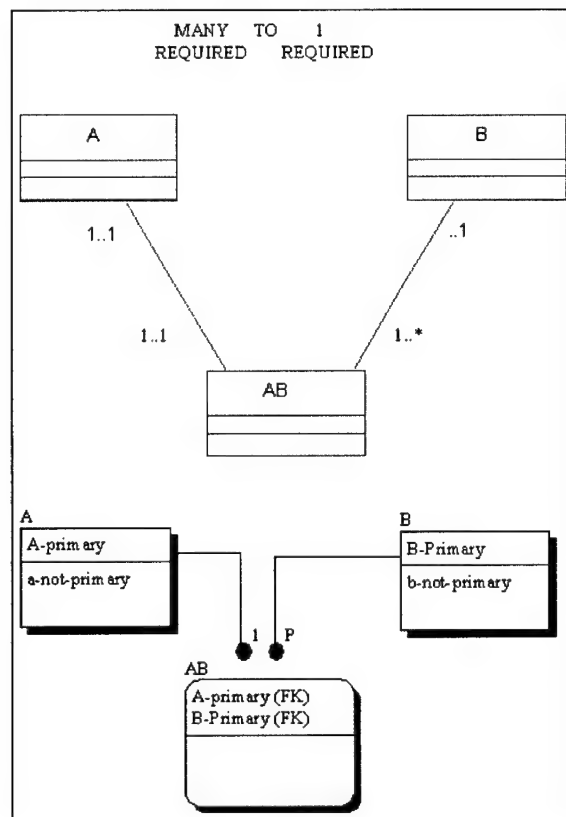


Figure 19. N-to-1 Both Mandatory Example Design by Associative Object

more, this method lacks association structure, and the application software must handle all management.

3.2.2.3 M-to-N Associations. Many-to-many associations are implemented as associative objects regardless of whether the object's participation is mandatory or optional. This is the only method that preserves third normal form. This method does not allow the possibility of circular data dependencies. Migrating a primary key from one object to another can cause duplication in the primary key unless the association is mandatory for both objects and the migrated key helps form a complex primary key. A single table can easily have duplications of key value pairs and circular data dependencies.

3.2.2.4 Aggregation. Each **has-part** relationship is handled individually according to the relationship's properties. These are determined by the cardinality of the relationship and whether the component object can exist alone or only as part of the aggregate. Aggregations often exhibit existence dependency in the component objects as **Frame** does in Figure 20. The implementation of these relationships, as with the associations defined earlier, requires designer input. These are often implemented by migrating a primary key from the composite object into the component object. This key must at least be part of the primary key for the component object if it is identity dependent. It is also acceptable to implement this association by creating an associative object. This is determined by the particular association and the designer.

3.2.2.5 Special Case: Link Attributes. Link attributes are attributes that are not part of either entity in an association, but rather, the association itself. The link attributes can occur in 1-to-1, N-to-1, and M-to-N associations. In this research, an association having a link attribute must be declared as an associative object in the specification. This is the preferred method of implementation, and associations do not have attributes in the AWSOME structure. However, there are two methods that can handle link attributes in an association in general.

The associative object method, Figure 21, can be used for 1-to-1, N-to-1, and M-to-N associations. This object is created by taking the primary keys from all the objects in the

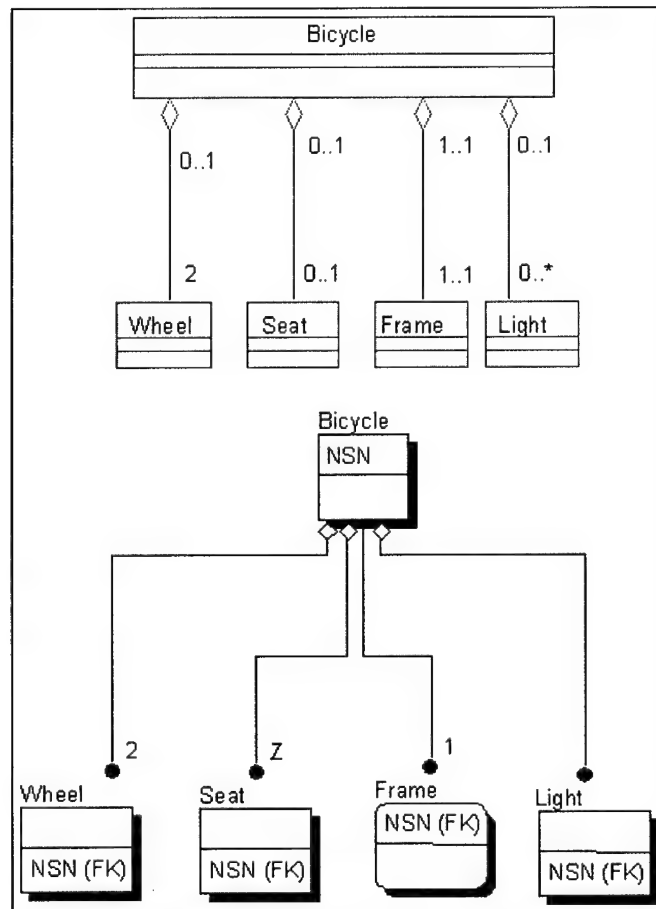


Figure 20. Aggregation Implementation Example

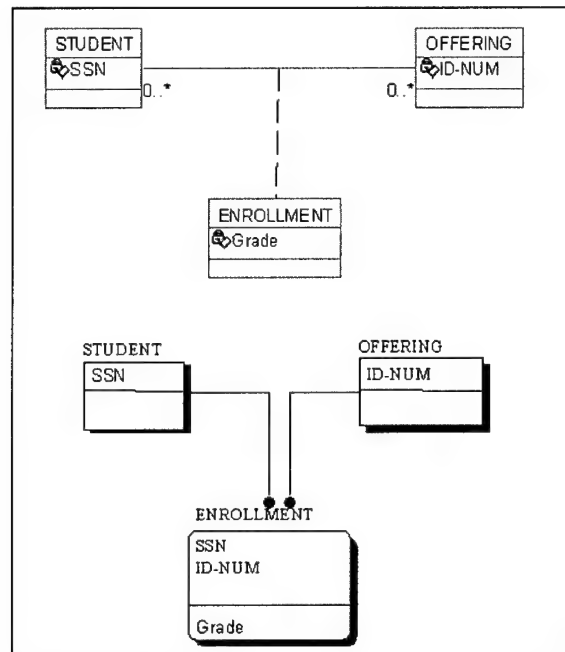


Figure 21. Associative Object Link Attribute Implementation

association to form a complex primary key for the new object. The link attributes can then be placed within the associative object as normal object attributes. Each instance of the associative object represents one instance of the association between the primary objects. Searches for link attributes are performed by a single search in the associative object table. Additionally, this table promotes easy association retrieval by searching the associative object and then using the results to search the primary tables. This method requires additional overhead as a third table must be maintained. Consequently, all association queries requiring more than the primary keys or link attributes will require at least two table lookups. This structure reveals the nature of the link attributes by the table construction. Associative object is the preferred choice for this relationship. This method gives more inference of the relationship from the table structure. It clearly shows the link attributes as related to both objects.

Another method migrates a primary and embeds the link attribute in one of the objects of the association. In all three cardinality situations, the link attribute is placed in the object that receives the migrated key. The link attribute field must be nullable since it will not have a value unless the association exists. This method hides the fact that the

link attribute is related to the association. Instead, it identifies the attribute as a member of the object in which it is placed. This method breaks normal forms, and when queried, the object receiving the foreign key must be known.

An additional method is to use a single table to represent both primary tables and the association. It embeds the link attribute of the association and both objects into one object. The single table method requires the least code to create. However, it lacks association structure and the application software would handle all management. This method breaks first normal form for the M-to-N and possibly in the N-to-1 associations. This could be corrected by using a unique, system-generated primary key in these cases. However, in all cases second normal form would be broken. The table's structure does not imply any information concerning the association.

If a design decision other than associative object is wanted, embedded link attribute is the second choice, however poor it may be. This method loses the ability to infer information about the original object-oriented association from the table structure. Also, the loss of normal forms and the additional overhead of determining which object contains the link attributes make this method a poor alternative.

Single Joint Table is the least preferred choice. This method works best when the association is very dense, and takes the least code to implement. This method is a poor choice because it does not conform with lower normal forms. It drops the individuality of the objects involved and the structure of the link attributes.

3.2.2.6 Special Case: Qualifiers. A qualified association, as shown in Figure 22, relates two object classes and a qualifier. The qualifier is an attribute that reduces the effective multiplicity of an association in M-to-N and N-to-1. For N-to-1 associations, the qualifier is placed on the N cardinality object.

Associative Object is the only acceptable method used in this effort for a qualified association. The primary keys of both objects and the qualifier form the new associative object's complex primary key. This method shows the identity dependence of the association upon the qualifier. Also, this method retains third normal form for the association relation and the structure of the table shows the qualified association.

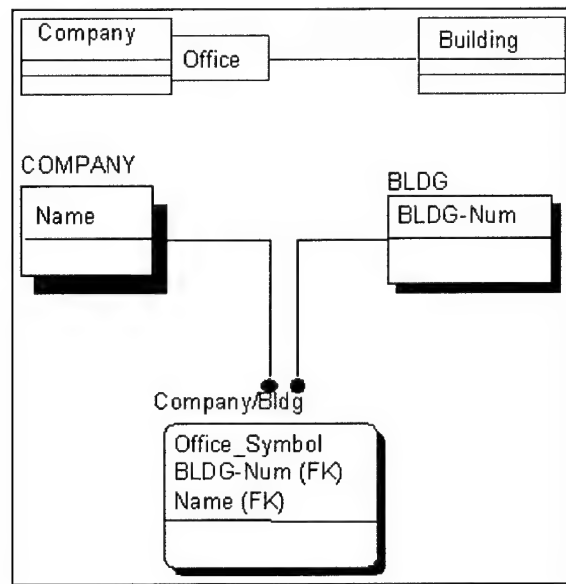


Figure 22. Qualifier Implementation

3.2.2.7 N-ary: Ternary and Higher Order Associations. These associations may be of different degrees. They are rare and are handled as an oddity. Associations may be binary, ternary, or of higher order. In practice, the vast majority are binary or qualified. Higher order associations are more complicated to draw, implement, and think about than binary associations and should be avoided if possible. Many associations between three or more classes can be decomposed into binary associations or phrased as qualified associations. If a term in a ternary association is purely descriptive and has no features of its own, it is a link attribute of a binary association. An example of a ternary association is “Quarterback plays for team during year”. This association cannot be decomposed without losing information.

Since ternary and higher order associations are encountered so infrequently, only one method for transforming them is recommended in this research. This method, as shown in Figure 23, is to implement the association as an associative object. It captures the association structure in the database as the dependent object’s table receives a portion of its primary key from each of the other tables involved in the association.

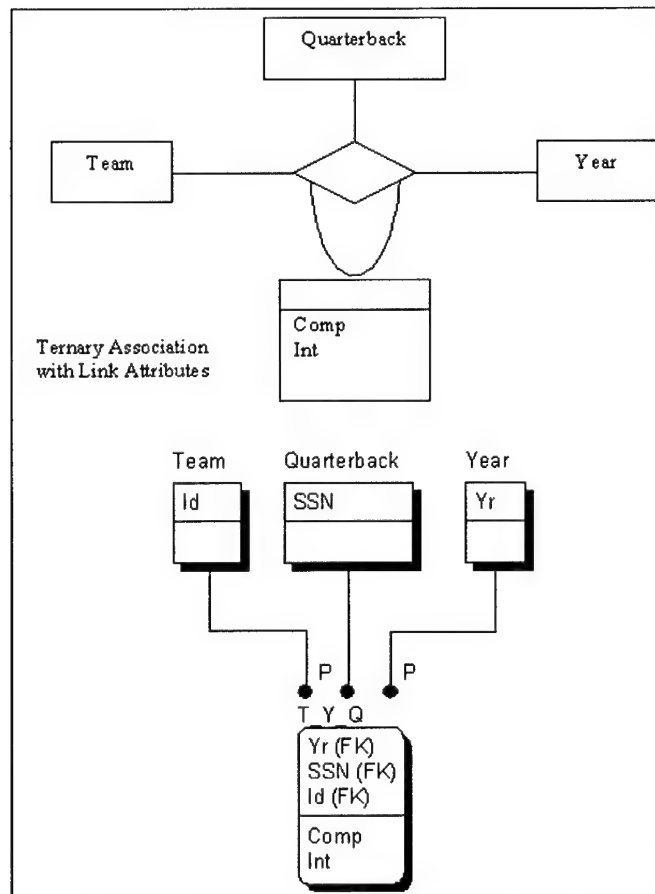


Figure 23. Ternary Association Implementation

3.2.3 Transformation of Associations. Associations can be implemented in a number of different ways. Based upon the analysis in the previous section, two methods were found to be the most versatile and have the fewest negative side effects. For this research effort, only those two methods, the creation of an associative object or the migrating of a primary key, were allowed. Once a primary key is migrated, it may be assigned as a part of or as the primary key for the receiving class. It is left to the designer not to change the meaning of the association. Although it is strongly discouraged, for an N-to-1 association, the designer may migrate a primary key from the N-cardinality association end to the 1-cardinality association end. This foreign key is normally implemented the other way. In this case, it is again left to the designer to make an intelligent, informed decision concerning the design of the database.

In some cases, the association implementation decision is made by the system. Associations that have more than two participant objects are automatically transformed to associative objects. This is done by checking the number of elements in the `WsAssociation` element, `wsAssociationEnd`. Additionally, if the association has a cardinality of M-to-N, the association is automatically transformed into an associative object, Figure 24. Any association not transformed by the first check of degree is now assumed to be a binary association. In this case, the element, `wsAssocEndMultiplicity`, in each association end is checked. Each `WsIntegerType` in the sequence for a given end is checked. If the highest upper bound for an association end is greater than 1, the cardinality of that association end is treated as N cardinality for this check. If both ends have N cardinality, the association is transformed into an associative object.

An association that does not have cardinality greater than two or is not M-to-N, has its transformation directed by the designer. When the designer selects an association, the cardinality of the two association ends is given. The designer then decides to implement the association with an associative object or by migrating a primary key.

If the associative object method is selected, a new `WsAssocObject` is created. Both `WsAssocObject` and `WsAssociation` extend the abstract class `WsDeclaration`. The name of the association is concatenated with the string, `A0`, to name the new Associative Object. An example of this is the association, `Assigned`, that lends its name to the new associative

The association end Student is Optional with a cardinality of N		
The association end Section is Optional with a cardinality of N		
The association has a cardinality of N-to-N and must be handled as an associative object		
The association Assigned is implemented as the AssocObject: AssignedAO		
The Transformed Attributes of the Associative Object: AssignedAO		
<u>Primary Key</u>	<u>Name</u>	<u>HomeClass</u>
true	Student_Person_ssn	Student
true	Section_Offering_Course_ctype	Section
true	Section_Offering_Course_cnum	Section
true	Section_Offering_Quarter_qname	Section

Figure 24. M-to-N Association Transformed to Associative Object

object, **AssignedAO**. This naming convention makes it easy to recognize associative objects created as a result of an association transformation. The element, **wsAssocObjectEnd**, is set to the association's **wsAssociationEnd**. Finally, the association's element, **wsAssocImplement**, is set to the **WsIdentifierRef** that points to the new associative object's name. In this manner, the design decision for the association is recorded for later use in DML generation.

If migrating a primary key is the method selected as shown in Figure 25, the designer is reminded of the cardinality of the association ends. The designer then selects which class, referenced by the association ends, will migrate its primary key. The selected class has clones created from the attributes that make up its primary key. These attributes are given new names to ensure uniqueness. The new name consists of the association name, two underscores, the name of the class giving the key, a single underscore, and the original attribute name. This naming convention ensures that different associations between the same objects have unique names that act as a road map for the designer when trying later to determine the origin of a field. These are imported to the attribute list of the class referenced by the other association end. These attributes are not identified as part of the primary key for that table. The association's element, **wsAssocImplement**, is set to the

Association: MemberOf		
The association end Student is optional with a cardinality of N		
The association end GradClass is optional with a cardinality of 1		
The association MemberOf is implemented by passing a foreign key to Student		
The Attributes of the Class: Student		
<u>Primary Key</u>	<u>Name</u>	<u>HomeClass</u>
false	gpa null	
true	Person_ssn Person	
false	MemberOf__GradClass_designator	GradClass
false	RAdvises__Faculty_Person_ssn	Faculty

Figure 25. Association Implemented by Migrating Foreign Key

WsIdentifierRef that points to the class that received the migrated key. In this manner, the design decision for the association is recorded for later use in DML generation. Error checking is performed prior to implementation to ensure that the class giving the key has been transformed. If a class has not been transformed, the primary key cannot be imported. If the foreign key is to entirely compose or be part of the receiving class's primary key, the designer must transform that class to identify the primary key.

3.3 Associative Objects

Associative objects occur under two circumstances. The designer can choose to implement an association as an associative object, in which case the transformation of the newly created associative object occurs automatically. The other situation occurs when the associative object is declared in the specification. This can occur for numerous reasons. If an association has attributes, methods, and/or a qualifier, or it participates in another association, it must be declared in the specification as an associative object. Additionally, the designer may choose to represent an association as an associative object without any of the criteria listed above. WsAssocObject, Figure 26, represents the associative object

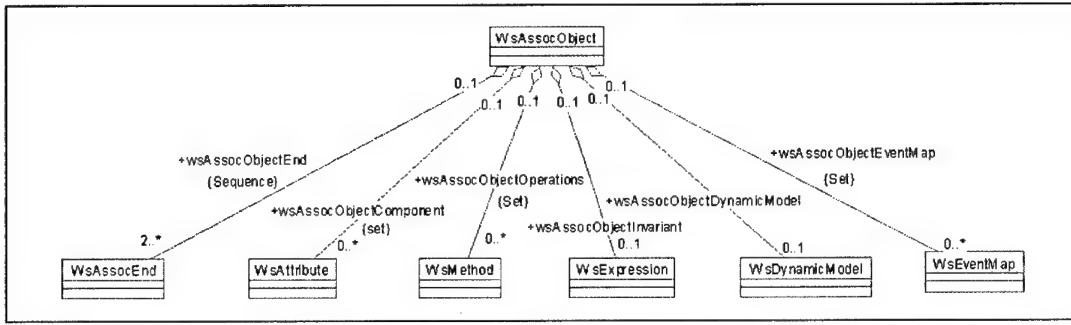


Figure 26. WsAssocObject Structure

in AWSOME. It has all the elements of a class except superclass and has the additional element of a sequence of association ends. The associative object will reference at least two identifying classes through the association ends. This is because an associative object has all the capabilities of both a class and an association. The associative object is assumed to be persistent and concrete in this research effort.

In order to transform the associative objects into structures from which DDL is generated, the instances of WsAssocObject are first identified to the designer. Each instance is then automatically transformed into a form from which DDL may be generated. These structures could be further transformed into classes. However, in order to maintain the design decisions of the process, this effort does not make this next transformation, but rather, generates the DDL from associative objects. By doing this, the associative object maintains all the functionality of both an association and a class.

This transformation has two steps. First, all the association ends of the associative object are examined to determine the class to which each particular association end refers. New attributes, clones of the attributes that make up each referenced class's primary key, are then imported to the attribute list of the associative object. These attributes are all identified as part of the primary key for the table that is to represent the associative object. This is shown in Figure 27. Error checking is performed to ensure the referenced classes have been transformed. If a class has not been transformed, the primary key cannot be imported. Additionally, the associative object is checked to ensure it has not already been transformed. This prevents the attempt to add an attribute that has already been imported.

The Transformed Attributes of the Associative Object: Offering		
<u>Primary Key</u>	<u>Name</u>	<u>HomeClass</u>
false	code	null
true	Course_ctype	Course
true	Course_cnum	Course
true	Quarter_qname	Quarter

Figure 27. Transformed Associative Object Without Qualifier

The second step is to determine whether there is a qualifier. For this effort, any association with a qualifier must be defined in the specification as an associative object. Additionally, the qualifier must be declared as an attribute of the associative object rather than an attribute of one of the classes referenced by association ends. To determine whether a qualifier is present, the association ends of the associative object are examined. If the association end element, `wsAssocEndQualifier` is not null, the `WsIdentifierRef` will indicate the `WsAttribute` of the associative object that is the qualifier of the association end. This attribute is located in the associative object's element, `wsAssocObjectComponent`, and the `WsAttribute`'s field, `wsPrimaryKey`, is set to true. This indicates that the selected attribute acts as an identifier and is placed as part of the primary key for database relation that is generated from the associative object. These two steps complete the transformation of associative objects for the generation of DDL.

3.4 Class/Associative Object Invariants

The use of class/associative object's invariants is necessary to avoid losing information during conversion to DDL. This is because various software languages and databases have differing capabilities in the use of invariants. In this research, the multiplicities of the associations are captured and placed within the class/associative object's invariants. This research does not utilize these invariants further, but rather, recognizes their value and preserves this information for future use.

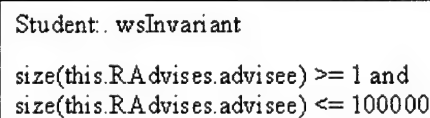
The multiplicity of an association, 1-to-1, 2-to-100, etc., whether expressed as an associative object or association, is recorded in the association ends under `WsAssocEnd`,

wsAssocEndMultiplicity. These multiplicity constraints are currently not used by the transform system other than to inform the designer of the association multiplicity and mandatory or optional nature during the design process. These constraints could be put to further use with additional development of this research.

Many database systems allow the use of triggers. These triggers define actions the database should take when some database-related event occurs. Triggers may be used to supplement declarative referential integrity, enforce business rules, or to audit changes to data. Triggers execute within the database when data manipulation commands such as inserts, updates, and deletes are performed. The capability to use triggers and the syntax for declaring them differ between database systems.

This effort does not currently implement triggers. However, it is realized that the constraint data is important and could be enforced by a specific database plug-in or in the software application itself. To capture this information, these multiplicity constraints are saved within the AWSOME structure during transformation. When an association is transformed by migrating a primary key, the multiplicity of each object's participation in the association is placed within each object's class/associative object constraint.

This constraint, as shown in Figure 28, uses the predefined AWSOME function, `size`. The argument passed to `size` is `this.Association Name.Role Name`. This argument states the inclusion of the current class in the association with its role name. This function call is then paired with the upper and lower multiplicity bounds of the `WsAssocEnd`, `wsAssocEndMultiplicity`. This preserves for future use the association multiplicity constraint within the classes that participate in the association.



```
Student.wsInvariant  
  
size(this.RAdvises.advisee) >= 1 and  
size(this.RAdvises.advisee) <= 100000
```

Figure 28. Class Invariant for Multiplicity Constraint of Association

3.5 DDL Generation From AWSOME

Once the specification has been parsed into AWSOME and the previously explained transformations are performed, DDL for creating the relational database structure can be generated automatically. This is performed by first checking all the classes, associative objects, and associations to ensure that they have been transformed. The designer will be informed of all objects that have not been transformed. The automated system cannot guarantee that the software system is designed in the most efficient manner or that the designer has not made mistakes as human decision is part of the design process. The system can determine whether any given class/associative object has a primary key and whether all associations have a referenced implementation.

If all objects are determined to be transformed, all classes and associative objects are checked for circular dependencies. If a situation exists where a number of objects create a circular dependency through the migrating of keys, the designer is informed of those offending objects and the process cannot continue until the errors are fixed. If there are no circular dependencies, each object will be examined to create a string that, when passed to the relational database, will create a corresponding table. These objects are each processed in an order that will not create a dependency error by attempting to reference a table that is yet to be generated. The components of the `WsAssocObjects` and `WsClasses` are read and interpreted to create a string. This string is then passed to the RDBMS to create a corresponding table. In this process, the name of the object is used as the name of the table. The name of each attribute is used as the name for a column in the table.

It is very important to ensure that the domain for a database's column matches the datatype of the AWSOME attribute. Errors in this transformation could cause the entire software system to fail. However, an issue exists that makes this conversion very challenging. Each database system uses a different set of native datatypes. These datatypes are basically the same; however, each database uses slightly different parameters and names to define them. For instance, Oracle uses `VARCHAR2()` to represent a string of fixed length. Sybase uses `CHAR()` and Microsoft Access uses `String`. This difference means that in a robust application, each supported database requires a different plug-in.

The AWSOME structure holds information that makes the conversion to any database possible. Strings can be represented in AWSOME as arrays of Char with a defined length. This gives the string length. Integer types in AWSOME have upper and lower bounds. These could be passed as column constraints for those databases accepting them. Another option is to check these constraints against the upper and lower bounds of database number types when column constraints are not allowed. Similar checks could be done for real/float and date types also. Due to differences among RDBMSs, the problem was avoided in this research by hardcoding the basic types defined in the School domain to those of the target database of the prototype, Oracle. Currently, the basic types, String, Char, Integer, Float, Date, and boolean, are checked and assigned according to Table 1. As previously mentioned, in a robust implementation, a separate plug-in would be needed for each database. Additionally, transforms would be needed to convert complex specification-defined types into native database datatypes.

Specification in AWSOME	Oracle Native Datatype
String	VARCHAR2(25)
Char	VARCHAR2(1)
Integer	INTEGER
Float	NUMBER
Date	DATE
boolean	VARCHAR(1)

Table 1. AWSOME to Relational Type Conversion

After type conversion, the next step in DDL generation is checking which attributes are part of the primary key. Those attributes making up the primary key are identified as NOT NULL. Additionally, each element is checked to see whether it is also part of a foreign key. All the foreign keys are then listed in table constraints indicating the table from which they came and the attribute they reference. Finally, the primary key for the table is identified. This string, as shown in Figure 29, is then sent to the database and the table is created.

These transformations on the specification parsed into the AWSOME structure provide the framework to generate DDL. By preserving the transform decisions, DML that will interface properly with the DDL produced from the specification can be generated.

```

CREATE TABLE Section (
  snumber                                INTEGER,
  TaughtAs__Offering_Course_cnum         INTEGER          NOT NULL,
  TaughtAs__Offering_Quarter_qname       VARCHAR2(25)    NOT NULL,
  Teaching_Faculty_Person_ssn            VARCHAR2(9),
  FOREIGN KEY
    (TaughtAs__Offering_Course_cnum ,
     TaughtAs__Offering_Quarter_qname)
    REFERENCES Offering(Course_cnum , Quarter_qname),
  FOREIGN KEY (Teaching_Faculty_Person_ssn)
    REFERENCES Faculty(Person_ssn),
  PRIMARY KEY
    (TaughtAs__Offering_Course_cnum ,
     TaughtAs__Offering_Quarter_qname))

```

Figure 29. DDL Code Automatically Generated From AWSOME

The importance of wise design decisions by the designer in the implementation of the associations cannot be overemphasized. An automated system can make recommendations or blanket decisions such as transforming all associations to associative objects. However, the best decisions still require human intervention as intent and personal preferences cannot always be specified in a logical manner. Embedded DML from the specification can only be generated if the design decisions are recorded.

IV. Generating Data Manipulation Language (DML)

In order to generate DML from a formal, object-oriented specification, one must know the design decisions used to build the RDBMS structure. Since the table structure alone does not reflect the inheritance, associations, and aggregation, one must also know the way these object-oriented entities are implemented. Chapter 3 describes the process of capturing the software system specification in the AWSOME AST structure. The steps for extracting the persistent data structure, transforming the object-oriented schema into a relational database schema, and preserving the design decisions are also described in detail in Chapter 3. After these actions are completed, it is possible to proceed with the second main step of designing an RDBMS, that is, the implementation of specified requirements or functions. Generating executable DML code to retrieve and manipulate data, as specified, does this. This research deals only with the retrieval of data. The manipulation of said data is left for the further development of this effort.

This chapter describes the process of recognizing database retrieval operations and implementing them in code. This is done by recognizing expression patterns, interpreting their content, and generating the mathematically equivalent database statements. The post condition of the operation is the part of interest. The specification for the post condition only describes the state that should exist after the operation occurs. It does not tell how this operation is to be executed. This research develops a methodology for determining how to correctly implement the post condition in the form of DML. Another important point is that SQL statements return sets of records. The set may only contain one tuple or none at all; however, it is still a set. Consequently, the post condition defines the return set. As a result, the methodology of this research determines how to return the specified set.

This chapter first describes how the post condition expressions used in the prototype of this research are represented in AWSOME. The process of pattern matching and its use to determine whether the operation can be converted to DML code is explored. In this conversion, the post condition expression is evaluated to determine what should be returned from the operation. The constraint that defines the data to be returned is interpreted. This information is then compiled and translated into DML. Finally, this chapter explains how

```

function getMyFaculty(InputQname : in String) : FacultySet is
    guarantees getMyFaculty(InputQname) =
        { f | ( f : Faculty, s : Section, o : Offering, q : Quarter )
              s in f.Teaching.Teaches and
              o in s.Taught_As.Theory and
              q in o.Offering.Time      and
              q.qname = InputQname      and
              this in o.Assigned.Assignee);

```

Figure 30. AWSOME Specification for Function: getMyFaculty()

the previously transformed inheritance and association requirements are used to ensure that the correct information is retrieved by the DML.

4.1 Representation of Expressions in AWSOME

Before the methodology for generating DML can be understood, one must have a knowledge of how methods, such as the one shown for the class `Student` in Figure 30, are represented in AWSOME. Since these methods must be used for the generation of application code as well as DML, all actions dealing with the specifications in this research are nondestructive in that they do not alter the structure but merely read it in its current form. An extension not pursued in this effort is the transformation of methods that are not in an interpretable form into a form that can be interpreted by the automated DML generator.

4.1.1 WsMethod. Both classes and associative objects can have methods. This is expressed in `WsClasses` by the component, `wsClassOperations`, and in `WsAssocObject` by the component, `wsAssocObjectOperations`. In both cases, these components are sets of `WsMethod` as shown in Figure 31. `WsMethod` has two boolean attributes and one component. The boolean attribute, `wsPrivate`, indicates whether the method is private or public. In the AWSOME structure, methods cannot be declared as protected. The boolean attribute, `wsClassMethod`, indicates whether the method is a class or instance method. This attribute is used in DML generation and will be discussed further in Section 4.2.2. Finally, the component, `wsMethodSubprogram`, is represented by the AWSOME structure `WsSubprogram`.

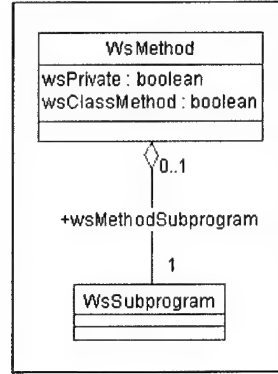


Figure 31. WsMethod Structure

4.1.2 WsSubprogram. WsSubprogram, Figure 32, extends WsDeclaration. As a result of this inheritance, `wsDeclName` is used as the name of the operation. WsSubprogram has a number of components that are used for DML generation. The component, `wsSubprogFormals`, is a sequence of WsParameter. These are the arguments passed to the operation. They may be classified as IN, OUT, or IN OUT parameters. `wsSubprogLocals` is a set of WsDataObject. This structure represents the local variables and constants declared within the operation. `wsPostConditions` is a WsExpression. The WsExpression hierarchy is by far the most complex structure within AWSOME and is used repetitively throughout the specification of methods. In fact, a large portion of the DML generation involves evaluation of this structure. `wsPreConditions` is also a WsExpression. However, in this research the precondition is assumed true and consequently is not used. The remaining component, `wsSubprogBody`, and the boolean attribute, `wsExternal`, are not used in DML generation. WsSubprogram is an abstract class that is implemented by WsProcedure and WsFunction. WsFunction has a return component, `wsFuncReturnType`, that is a WsName. WsProcedure does not have any components.

4.1.3 WsContainerFormer. There is an additional structure that must be described. WsContainerFormer, Figure 33, is used as part of the post condition for pattern matching. Specifically, it is the concrete extension of this abstract class, WsSetFormer, whose components define the set that the operation is to return and constrain the information within the system to only that which is desired. However, WsSetFormer does

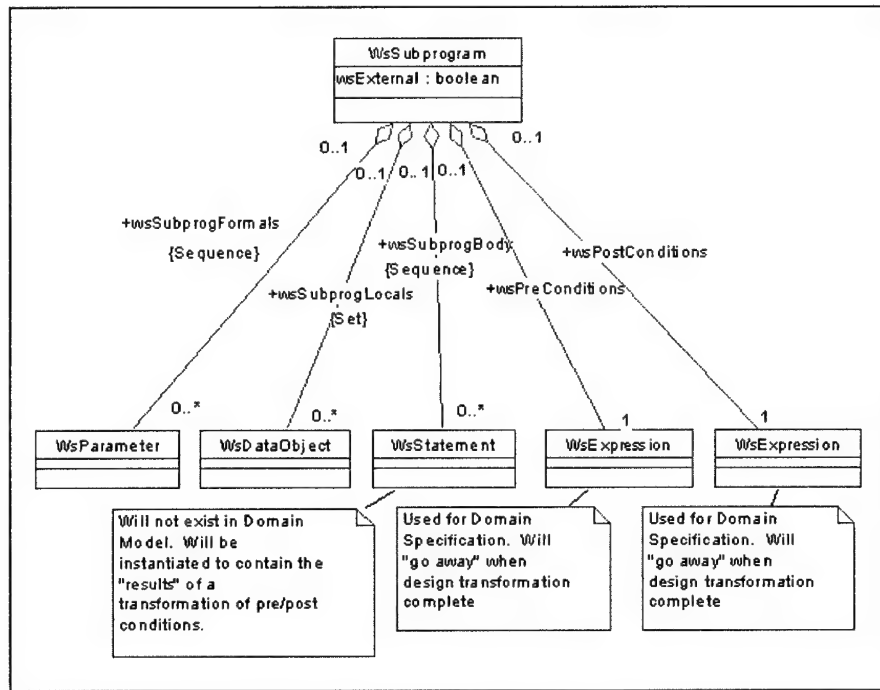


Figure 32. WsSubprogram Structure

not have any additional components or attributes. WsContainerFormer, which extends WsExpression, has three attributes, all of which are used within this research to generate DML.

wsContainerExpression is a WsExpression. This expression defines the items that make up the return set. For this research, this expression was limited to a class or an attribute of a class. Complex structures created with multiple attributes from a single class

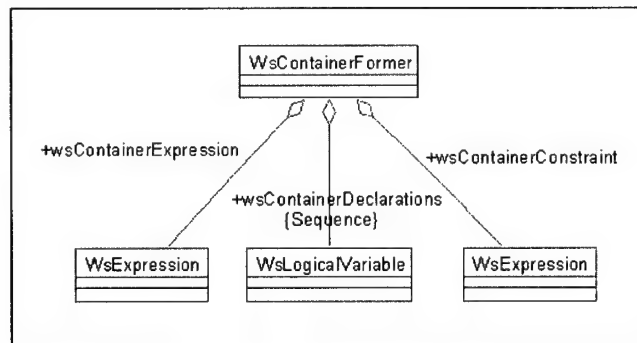


Figure 33. WsContainerFormer Structure

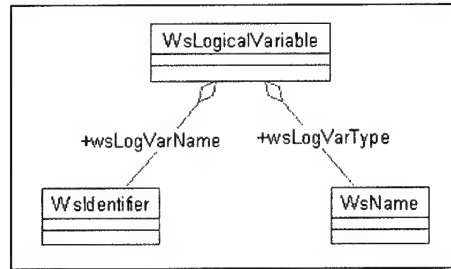


Figure 34. WsLogicalVariable Structure

or numerous classes were not implemented. To do so would require the definition of tuples made from the components that this research does return. Specification of complex tuples is possible within the AWSOME structure. For a more robust application, this research could be extended to include complex tuples. `wsContainerConstraint` is a `WsExpression`. This constraint provides the meat of the DML queries and is where the return set is constrained in terms of association participation and attribute values. It may contain logical and mathematical operations over multiple classes, associative objects, attributes, associations, variables, and constants.

`wsContainerDeclarations` is a sequence of `WsLogicalVariable`. `WsLogicalVariable`, as shown in Figure 34, has two components. `wsLogVarName` is a `WsIdentifier`. This is the name of the logical variable. `wsLogVarType` is a `WsName`. This is the data type of the logical variable. These logical variables are used only within the container former in both the container declaration and constraint.

4.1.4 Pattern Matching. The post condition of an operation in AWSOME is a `WsExpression`. As stated earlier, `WsExpression` is the most complex structure within AWSOME. In order to determine whether DML could be generated from a post condition, the structure of the post condition was evaluated to see whether it was in a format that could be recognized by the DML generation program. Developing transformations from unacceptable formats to acceptable formats was not part of this research. This effort only determined whether the post condition was in a format that could be evaluated. Since database queries for data, not including functions such as COUNT, MAX, MIN, etc., return a set, a standard condition format was used. This format of `WsSubprogramCall` =

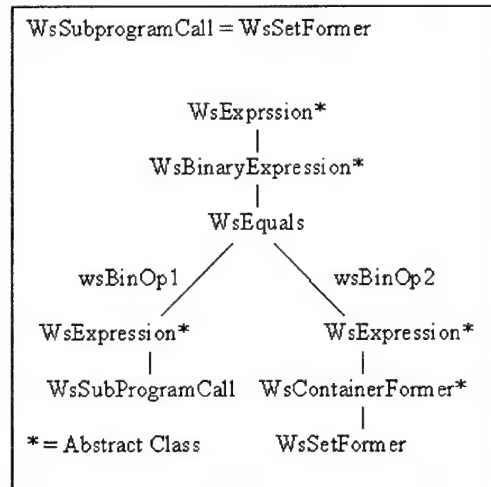


Figure 35. Pattern Matched in a Post Condition

WsSetFormer is shown in Figure 35. The **WsSubprogramCall** refers to the operation being defined, and the **WsSetFormer** describes the set that is returned by the operation.

Another pattern that is matched in DML generation is the **WsIn** structure within **WsSetFormer wsContainerConstraint**. Figure 36 shows the AWSOME representation of a **WsIn** statement with abstract inheritance levels in the diagram that the AST lacks. In this case, the **WsIn** is used to indicate the participation of objects within an association. When a **WsIn** is detected, the components are checked. **wsBinOp1** should be a **WsThis** or a **WsIdentifierRef** that points to a variable with a datatype of a class/associative object. **wsBinOp2** is a selected component. The selected component's element, **wsSelCompComponent**, should be a **WsIdentifierRef** that points to an association role name. The element, **wsSelCompName**, is another selected component. The second selected component's element, **wsSelCompComponent**, should be a **WsIdentifierRef** that points to an association. The second selected component's element, **wsSelCompName**, is a **WsThis** or a **WsIdentifierRef** that points to a variable with a datatype of a class/associative object. From this pattern, association participation can be interpreted. By using the design decisions retained from the DDL transforms, the corresponding DML substring that correctly reflects the specification can be generated.

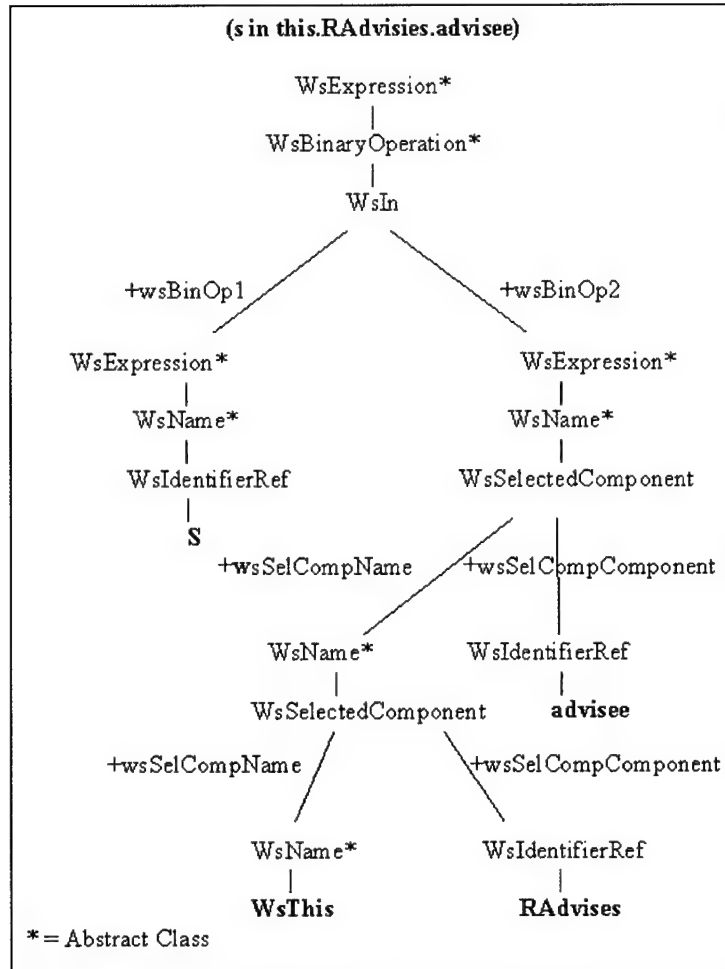


Figure 36. WsIn Association Representation Example

Pattern matching can be extended to other instances. Often certain components are expected, and the various options are given to match an executable pattern. When none of the known patterns can be matched to a portion of the specification, transforms must be created to adjust the specification to a mathematically equivalent form that can be matched to the pattern. Pattern matching is not perfect. In fact, there may be patterns that cannot be matched or a specification that cannot be transformed to a recognizable pattern. A simple example of this is $X = X' * X'$. In this case, we are looking for the square root of X . However, very few pattern matching programs would recognize this operation unless this pattern were specifically predefined as a special operation.

4.2 Generating DML From AWSOME

With an understanding of the AWSOME structure that holds the specification for methods to be transformed into DML and how patterns are used to determine whether an operation is transformable, one is equipped to discuss DML generation. To generate DML, each class and associative object is individually examined for the presence of methods. Each operation is then checked against the `WsSubprogramCall = WsSetFormer` pattern. Any operation meeting this requirement is ready for further examination.

The DML statement is broken into three fragments: a `select statement`, a `from statement`, and a `where clause`. The `select statement` tells the database what set of records to return. The `from statement` indicates which tables are to be used in the query. Finally, the `where clause` constrains the operation to only the data required. For this effort, these three statements are built in parallel and then concatenated when finished to form a single DML statement for each operation. There are numerous other functions that could be included such as group by, order by, max, min, count, etc. This research uses a limited implementation and does not contain all SQL functions. Rather, it is limited to simple queries. These queries may not be optimized for query execution. However, most RDBMSs have an internal query optimizer. In addition, there are COTS products such as those discussed in Chapter 2 that are expressly designed to optimize SQL queries. This research does produce mathematically correct queries. Further developments of the SQL functions and query optimization are left to later expansions of this research.

4.2.1 Runtime Variables. There are numerous variables that are not known until the application code is run. Passed parameters and local variables, whose values depend upon other input and the attributes of the instance that calls the operation are all unknown until time of execution. Respective RDBMSs handle these variables in different manners and syntaxes. Since there is no one way to represent these situations correctly, a different package with this information is needed for each RDBMS. To handle this situation, any time an instance attribute is called for, it will be represented in the DML string by the form, `this.attribute name`. This method identifies the instance, and the attribute name indicates the attribute whose value would be used. In a case where the variable is a formal argument or a local variable to the operation, its existence is represented by the variable name. Another option is to leave a placeholder in the DML statement and to also provide a vector of the runtime variables. However, to make the code more readable and more easily understood for academic purposes, the first method described was chosen for this research.

4.2.2 Class and Instance Methods. The `WsMethod` boolean attribute, `wsClassMethod`, is checked to determine if the operation is an instance or class method. If it is a class method, no action is taken. However, if it is an instance method, the instance is associated with the DML `where` clause. This is done by adding the instance's primary key to the return set constraint as expressed in the `where` clause. The form of this statement is `this.primary key = object.primary key`. The string `this.primary key` is the primary key field of the instance calling the operation. The string `object.primary key` is the name of the object containing the operation that corresponds to the table name and the name of the attribute that is the primary key. If the primary key is a composite key, then the individual statements for each part of the key are `anded` together. In this manner, the instance is included in the return set constraint.

4.2.3 Determining the Return Set. There are two locations that specify the return set in the operation specification. In the operation declaration, if it is a function, there is a return type declared. However, the datatype could be defined in basic types such as string or integer. It might not necessarily reflect the actual fields being returned such as

the `ssn` attribute of the class `Person`. This would be even more evident if this work were extended to include complex records with fields from different tables. The other option is to use the set former to determine the actual make-up of the return set by the actual classes and attributes. This is the option taken in this research.

Once the post condition has been checked to determine that it is in the correct form, the `WsSetFormer` is examined. The component, `wsContainerExpression`, is captured. This field points to a logical variable that is represented in AWSOME by the structure `WsLogicalVariable`. Once the logical variable has been located, it is evaluated to determine its datatype. For this research, the datatype is limited to a class/associative object or an attribute of a class/associative object in the form `object.attribute`.

If the datatype is a class or an associative object, the object is found and its attributes are added to the `select statement`. The name of the object is then added to the `from statement`. If this object is a class that extends a superclass, it is involved in an inheritance hierarchy. In order to return the entire object, including inherited attributes, the superclass is recursively examined until a level is reached that has no superclass. At each level, the superclass's attributes are added to the `select statement` and the `from statement` receives the name of the object. The primary key/foreign key relationship defined during DDL transformations is added to the `where clause` in the form, `superclass primary key = subclass foreign key from superclass`. An example of this is `Person.ssn = Student.Person_ssn`. If at any level of this recursive procedure an associative object is encountered, the process ends at that level because an associative object cannot have a superclass.

When the datatype is an `object.attribute`, a similar procedure is followed. At each level of the superclass hierarchy between the object listed in the specification and the class that holds the attribute, each object's name is added to the `from statement`, and the tables are linked through the primary key to foreign key relationship in the `where clause`. The superclass object and attribute are added to the `select statement`. Once this process is finished, the `select statement` is complete. The three strings are returned and the DML generator is ready to examine the set former constraint.

4.2.4 Expressing the Set Former Constraint. In the `WsSetFormer`, the component `wsContainerConstraint` is a `WsExpression`. This expression can be very complex and have many components connected by the structures, `WsAnd` and `WsOr`. To process this expression, this research recursively broke it down into smaller pieces, translated them individually, and then combined them to create the completed `from` statement and `where` clause. The purpose of this implementation was to break the expression into atomic parts that were simple and interpretable. This is a form of pattern recognition that breaks a complex statement into portions the patterns of which can be accepted.

4.2.4.1 Binary Expressions. `WsBinaryOperation` is an abstract class that extends `WsExpression`. A binary operation has two components, `wsBinOp1` and `wsBinOp2`, both of which are `WsExpressions`. This class is instantiated by numerous concrete classes. The following are implemented in this research: `WsAnd`, `WsOr`, `WsGreaterThan`, `WsLessThan`, `WsGreaterThanOrEqual`, `WsLessThanOrEqual`, `WsEqual`, `WsNotEqual`, `WsAddition`, `WsSubtraction`, `WsMultiplication`, and `WsDivision`. For each of these expressions, parentheses are placed around the resulting DML code and the component expressions are recursively evaluated. `WsIn` is a binary operation that is implemented in this research. However, it is handled in a different manner and will be discussed separately in Section 4.3.4.3.

4.2.4.2 Unary Expressions. There are two unary expressions in the AW-SOME structure, `WsMinus` and `WsNot`. Both have a component of type `WsExpression`. The unary minus sign and the key word `not` are both common in RDBMSs and are translated directly into the `where` clause with the component expression placed in parentheses. The component expression is then recursively evaluated.

4.2.4.3 WsIn Expression. The `WsIn` expression is treated as a special pattern that is checked for format as described in Section 4.2. After the pattern has been confirmed, the components are evaluated to determine whether they express participation in an association. The `WsIn` has the following format, `op1 in op2` where `op2 = object.association.role name`. Next, `Op1` and the object position of `op2` are eval-

uated. They will be the pointers to the objects involved in the association and will be represented by `WsThis` and/or `WsIdentifierRef`. If the `WsThis` is present, it indicates the instance calling the method. The `WsIdentifierRef` reflects a variable whose datatype will be a class or associative object. The association and role name are also captured. The named objects are then checked to determine whether they participate in the association indicated. If an object is not present in the association as specified, then its superclass structure is recursively checked until the exact location of the association participation is located. During this process, at each level, the `from statement` is checked to see if the object name is already listed. If not, it is added. The `where clause` is checked to determine whether the superclass primary key to subclass foreign key relation is already represented. If not, it is added. After the participation in the association is ascertained, the DDL implementation of the association is evaluated. If the association is implemented by a foreign key, the primary key of the giving object table and the corresponding foreign key are placed in an equality statement which is added to the `where clause` if it is not already present. An example of this is shown in Figure 37.

If the association is implemented as an associative object, the `from statement` is checked to see whether it includes the participating objects' names and the name of the associative object. If any of the names are not present, they are added to the statement. Next, the objects that participate in the association are linked by their foreign keys to the associative object and this statement is added to the `where clause` if it is not already present. An example of this is shown in Figure 38.

4.2.4.4 Extensions of *WsName*. `WsName` is an abstract class that extends `WsExpression`. It has three concrete instantiations that are encountered in the recursive evaluation of the set former constraint. Within this research, these three structures are at

```
s in this.RAdvises.advisee
where ...
    Faculty.Person_ssn = Student.RAdvises__Faculty_Person_ssn
```

Figure 37. DML for an Association Implemented as a Foreign Key

```
this in o.assigned.assignee
```

```
where ...
```

```
this.Person.ssn = AssignedAO.Student_Person_ssn and  
AssignedAO.Offering_Section_Course_cnum = Section.Offering_Course_cnum and  
AssignedAO.Offering_Section_Quarter_Qname = Section.Offering_Quarter_qname
```

Figure 38. DML for an Association Implemented as an Associative Object

the bottom of the recursion. They are converted directly to portions of the DML code. These instances are in relational statements as shown in Figure 39.

WsThis is only present in instance methods. It is handled by printing the object's name in the **where** clause. If it is not already present, the object's name will be added to the **from** statement. WsSelectedComponent is a selected component that is not part of nor does not contain another selected component. In this case, the **wsSelCompName** will be a WsName that references an associative object or class. **wsSelCompComponent** is a WsName that references an attribute of the object from **wsSelCompName**. The string, **object name.attribute name**, is added to the **where** clause. If the object name is not already present, it is added to the **from** statement. WsIdentifierRef is a structure that points to a logical variable, formal variable, local variable, or an object of type class or associative object. If the WsIdentifierRef indicates a logical variable, its datatype should be a class or associative object the name of which is printed to the **where** clause and then to the **from** statement if it is not already present. If the WsIdentifierRef indicates a formal or local variable, the variable's name is printed in the **where** clause. If the WsIdentifierRef points to an associative object or a class, the object's name is printed in the **where** clause and then in the **from** statement if it is not already present. These three structures are only recognized by themselves in the instances described. If other

```
q.qname = InputVariable
```

```
where ...
```

```
Quarter.qname = InputVariable
```

Figure 39. DML for Selected Component and Input Variable

values are present, the pattern is not recognized, and a DML string cannot be created for this operation.

4.2.5 Aggregation. Aggregation was implemented in the DDL portion of this research. However, it was not implemented in the DML portion. Traditionally, applications that use RDBMSs for persistent data management do not contain large amounts of aggregation. The inability to handle aggregation well is often considered a weakness of RDBMSs. This is partially due to the unknown format of the return set and the large number of possibilities it could contain when querying an object with aggregate components. An aggregation might have optional parts present in one instance and not in another. In strong typed languages, the return set's structure needs to be known because a type needs to be built for each possible return. Another alternative is for the entire return set to be treated as a string and then parsed into proper components. The representation of aggregation implemented in this research does not allow all parts of an object to be found if the structure of an aggregate object has loops. For instance, if an object has aggregate components and one of these components has a superclass that links back into the inheritance tree of the original object, the recursive calls used to create the DML string could get into an infinite loop. This structure would need to be implemented by a different method such as a tree. Due to these difficulties, the representation of aggregation in DML is left to be an extension of this research.

4.3 Summary of DML Generation

This research shows the ability to generate DML from formal specifications for specific cases where the post condition expression is in an interpretable format. As more patterns and transformations to those patterns are added, the set of DML statements that can be generated will increase. Some issues such as the handling of aggregation and difference of datatypes and interfaces for different RDBMSs will still need to be resolved.

V. Results, Conclusions and Recommendations

This research effort began with the primary objective of automatically generating the structure to support software system persistent data and the embedded code to access that data through the use of a relational database system. The hypothesis proposed from the beginning and pursued throughout this research was that if the design decisions for converting an object-oriented schema to a relational schema implemented in DDL could be captured, those design decisions could be used to automatically generate the corresponding DML operations as specified.

To support this endeavor, object-oriented associations and techniques for representing the relational equivalent were examined. Once a set of transformations was developed, a structure was needed to capture the initial object-oriented specification and the design decisions made during transformation. This significant effort focused on the AWSOME structure with specific attention directed to classes, associations, and associative objects. Once the structure and transforms capable of producing a relational schema in executable DDL were defined, the research turned to DML generation.

Two challenges were immediately encountered: how to represent operation requirements in the AWSOME structure as pre and post conditions and how to comprehend those specifications so that corresponding DML code could be generated. Use of the set former notation to define the result set of queries solved the problem of specifying query results. Expressing post conditions as set formers is a fundamental technique that is quite common in data-intensive applications. Pattern matching techniques were used to limit the scope of possible expression formats and to facilitate the interpretation of post condition specifications. With all this in place, DML generation became a task of transforming specifications in set former notation and using the design decisions to determine relational schema negotiation. The result of this effort is a process for relational database design with corresponding data access code from a formal object-oriented specification.

5.1 Results

First and foremost, this endeavor demonstrated the feasibility of a means for generating DDL and DML for a software system from that system's specification. Other significant accomplishments worthy of mention were achieved. Additionally, numerous unforeseen obstacles outside the scope of this research were identified. These issues are also presented in this section.

5.1.1 Accomplishments. In light of the differences between associations and relations, one of the major challenges in this research was to represent the OO associations as relations. To do this, a thorough examination of different implementation methods was documented. The effects of different methods were compared to the intent of the association and the side effects that occur when it is transformed to a relation. This evaluation provided insight into individual transformations and the need to model associations as a separate structure that exists at the same level as classes.

This analysis was implemented as three groups of transforms on the specification as represented in AWSOME. These transformations were categorized as follows: the classes were mapped to tables; the associative objects were mapped to tables; and the associations were designed into database relations. For all of these transformations, additional information was required from the designer. The design decisions of all three categories of transforms were captured and retained for use in DML generation.

Methods were implemented for transforming inheritance and associations that capture the design decisions. These decisions were used to generate DML. For inherited attributes, the DML was automatically generated to link subclass attributes with superclass attributes throughout the entire inheritance hierarchy. Transform decisions for associations were used to implement association participation between objects. If a superclass of an object participated in an association, the DML was automatically generated to allow the subclasses to also participate in the association. This allowed data to be retrieved by traversing associations between objects and allowed objects to inherit associations as well as attributes.

This research resulted in a prototype application that fills the gap left by COTS products. By combining the ability to generate both DDL and DML from a specification, the DML operations can be guaranteed to meet the specified requirements. The prototype application is capable of generating DML query strings that return either entire records or specified fields from a table. The specification for a method must be represented as a set former structure. However, the constraint within the set former can be complex as it may be composed of multiple associations, classes, associative objects, variables, attributes, and constants. Additionally, a number of logical and mathematical operations over these objects are supported.

5.1.2 Obstacles. Numerous issues outside the scope of this effort were encountered. One issue, not fully explored, was the evaluation of complex operation post conditions. Pattern matching was used to recognize pieces of the post conditions. If an expression could not be evaluated, it was recursively broken down until the pieces could be recognized. However, this was done for a limited number of situations. The complexity of expressions within AWSOME did not allow full pattern recognition to occur. As a result, operations such as updates, inserts, deletes, and some types of complex queries could not be readily interpreted.

An assumption of this research was that post conditions were in a format that could be interpreted. This was a prohibitive constraint that limited the operations performed. This area requires more attention. Specifically, a series of mathematically equivalent transforms of the expression needs to be developed. If expressions could be transformed into a more easily interpreted form, DML generation capabilities would be greatly enhanced.

Aggregation was flattened and represented in DDL. However, this process needs further research to implement a robust solution for DML. The possibility of loops, circular referential dependencies in the data structure, was beyond the capabilities that the recursive pattern matching used to generate DML. This was because aggregation is logically a tree structure that in this research was being stored as a tuple. Since an aggregate object may have a varied structure due to optional parts, the composition of the return set of a query could not be retrieved in a simple query.

The issue of datatypes also needs further research. Although the AWSOME structure is capable of holding the specifications of user-defined datatypes and subtypes, application languages and RDBMSs have different syntaxes and native datatypes. Inconsistencies in this area prohibit a universal solution for output to all databases and languages. Also, the handling of return sets by application languages is considerably different depending on the strong typing requirements of the language. Transforms are needed to convert complex specification-defined types into native database datatypes. A possible solution would be to have different packages of special type conversion functions for different languages and RDBMSs.

This research produced a DML string for each operation. However, a way to embed the DML back into the AWSOME structure or a target language was not addressed. Additionally, variables were represented by printing their names into the DML. Differences in application languages limit solutions to these deficiencies. One possible solution would be to save the DML in a tree structure rather than a string. This would allow easier code manipulation for different languages.

5.2 Conclusions

It is possible to generate DDL and corresponding DML from specification. However, DML generation requires additional research to be fully implemented. Before complex data retrievals or operations requiring multiple queries can be performed, a number of obstacles must be overcome. Specifically, a way must be devised to handle aggregation when object structures contain loops. Also, additional patterns will need to be identified and transforms created to convert expressions into interpretable forms.

The majority of challenges encountered dealt with matching object-oriented concepts and capabilities to relational schema. Since object-oriented objects attempt to model complex, real-world objects, expression of these objects and their properties relationally requires a flattening or simplification of the data structure. This creates a problem in that this simplification can produce a large and equally complex series of tables; yet this is still not enough to fully accomplish the task.

OO associations cannot be expressed as relations without the use of triggers within the relational database to help enforce referential integrity, multiplicities, and other constraints. Foreign keys can be used to provide paths for tracking inheritance and traversing associations. However, the relational structure alone does not define and enforce these OO concepts. Application software and triggers are needed to fully enforce specification compliance. The relational structure can provide a map for gathering data and providing operations on that data. However, this structure can become very complex as tables become large with overhead from many complex foreign keys and the number of tables increases from implementing associations as associative objects.

Until this research is extended to solve the problems identified, a robust industrial strength DDL/DML generator will not be possible. In the interim, if the capability to generate OO application code exists, it is recommended that an object-oriented database be used. In this case, classes need only be declared as persistent, and the OODBMS will take care of the management that must be explicitly built into relational database. This solution bypasses the problems associated in matching OO and relational schema.

5.3 Recommendations for Future Work

Relational database systems are matter-of-fact for the foreseeable future due to existing applications and monetary investment. As a result, there are numerous areas in which this research could be extended. This research was specifically limited to recognizing a small number of expression patterns and performing only simple queries. A direct expansion of this effort would greatly enhance the automated OO to relational matching. Particular attention should be directed to three main focus points: aggregation, enhanced operational capability, and datatypes. Transformation of aggregation is required to create a robust OO to relational application. Current research leaves two problems to be solved in this area. The relational structure that represents an aggregation may contain loops not interpretable by this research. A methodology needs to be devised to either traverse these loops or represent aggregation in a manner that eliminates them altogether. The results of aggregation queries are random in that the result set may vary in composition for any

given object with optional aggregate components. Some strong typed languages cannot handle this situation.

Another limitation of this research was the confined query capability that was developed. In order to support an object-oriented application with a relational database, full data management capabilities are required. This includes inserting, deleting, and updating records as well as providing complex query capabilities. In order to accomplish these tasks, numerous areas need to be explored. Methods for representing these requirements need to be developed. Since the AST representing expressions is so complex, transforms to the specifications are needed to represent these requirements in a manner that can be converted to DML. The pattern matching of expressions should be expanded. The limited format used works only for simple query operations. It does not handle updates, inserts, or deletes of object instances or association participation.

Since different languages and databases use different and sometimes unique datatypes, additional research could develop a methodology for identifying and converting datatypes between different languages and databases. It should include conversion of simple datatypes between applications as well as conversion of complex datatypes to simple ones. The differences in datatypes between relational databases and the complex, user-defined datatypes available in OO languages are significant. Research in this area would solve a problem that requires direct human conversions and work-arounds to make systems interoperate.

Triggers are required to implement referential integrity, business rules, and constraints, and to perform operations within the database system. A significant area still to be developed is the identification of requirements for triggers. Based on such analysis, a methodology could be developed for building and implementing triggers on different systems to enforce integrity and other constraints as well as to perform operations required by the software specification. This would greatly enhance the automated OO to relational conversion.

5.4 *Summary*

This research effort provides a solid foundation for representing and transforming object-oriented specifications into relational database DDL and DML. Although there are still hurdles to be overcome, the ability to design application code and persistent relational storage in a single process rather than two separate and distinct undertakings is an achievable goal for which a need will exist as long as object-oriented applications are paired with relational databases.

Appendix A. School Specification in AWSOME Syntax Before DDL Transformation

This appendix contains a sample specification written in AWSOME syntax. The specification is represented in its original form before any DDL transformations are performed. This example contains declarations of types, classes, associative objects, and associations. The classes contain both attributes and methods. Figure 5 in Chapter 3 is a graphic representation of the structure of this example.

```
type zeroToN    is range 0      to 100000;
type oneToN     is range 1      to 100000;
type zeroTo1    is range 0      to 1;
type oneTo1     is range 1      to 1;
type Integer    is range -100000 to 100000;
```

```
type Date       is array [1..9] of Char;
```

```
type StringSet  is Set of String;
type FacultySet is Set of Faculty;
type StudentSet is Set of Student;
type BicycleSet is Set of Bicycle;
```

```
Class GradClass is
  var program      : String;
  var year         : Integer;
  var graddate     : Date;
  var designator   : String;
end Class;
```

```
Class Quarter is
  var qname        : String;
  var year         : Integer;
  var qstart       : Date;
  var qend         : Date;
end Class;
```

```

Class Course is
  var ctype          : String;
  var cnum           : String;
  var ctitile        : String;
  var cdesc          : String;
  var creditHours    : Integer;
  var lectureHours   : Integer;
  var abetDes        : Integer;
  var abetSci         : Integer;
  var abetMath        : Integer;
  var abetOther       : Integer;
end Class;

```

```

Class Section is
  var snumber        : Integer;
end Class;

```

```

Class Bicycle is
  var bikeSerialNum   : String;
end Class;

```

```

Class Person is
  var lastname        : String;
  var firstname       : String;
  var initial         : String;
  var birthdate       : String;
  var ssn             : String;
  var height          : Integer;
  var weight          : Integer;
end Class;

```

```

Class Faculty is Person with
    var academicRank      : String;

    function getStudentsAdvised() : StudentSet is

        guarantees getStudentAdvised =
            {s | ( s : Student) s in this.RAdvises.Advisee };

    function getStudentsAdvisedSSN () : StringSet is

        guarantees getStudentsAdvisedSSN =
            { s | ( s : Student.ssn) s in this.RAdvises.Advisee };

    function getMyBikes () : BikeSet is

        guarantees getMyBikes =
            { b | ( b : Bicycle) b in this.Owns.OwnedBy };

end Class;

Class Student is Person with
    var gpa                : Integer;

    function  getMyFaculty(InputQname : in String) : FacultySet is

        guarantees getMyFaculty(InputQname) =
            { f | ( f : Faculty, s : Section, o : Offering, q : Quarter )
                s in f.Teaching.Teaches and
                o in s.Taught_As.Theory and
                q in o.Offering.Time      and
                q.qname = InputQname      and
                this in o.Assigned.Assignee};

end Class;

AssocObject Offering is
    var snum                : Integer;

    role Program             : Course      is zeroToN;
    role Time                : Quarter     is zeroToN;
end AssocObject;

```



```

Association Assigned is
  role Assignee      : Student      is oneTo1;
  role AssignedTo    : Section      is zeroToN;
end Association;

```

```

Association MemberOf is
  role isIn          : Student      is zeroToN;
  role Contains      : GradClass    is zeroTo1;
end Association;

```

```

Association RAdvised is
  role Advisor       : Faculty      is oneTo1;
  role Advisee       : Student      is zeroToN;
end Association;

```

```

Association TaughtAs is
  role Actual        : Section      is oneToN;
  role Theory        : Offering     is zeroTo1;
end Association;

```

```

Association Teaching is
  role Teaches       : Faculty      is oneTo1;
  role TaughtBy      : Section      is zeroToN;
end Association;

```

```

Association Owns is
  role Owner         : Faculty      is zeroTo1;
  role OwnedBy       : Bicycle      is zeroToN;
end Association;

```

Appendix B. School Specification in AWSOME Syntax After DDL Transformation

This appendix contains the specification from Appendix A after DDL transformations in AWSOME syntax. All classes and associative objects have attributes assigned as primary keys. The association declarations have been designed as associative objects and as attributes passed from one class to another as foreign keys. The manner that the association was implemented is indicated in the association structure. Additionally, the multiplicities of the associations are represented as class/associative object invariants.

```
type zeroToN    is range 0      to 100000;
type oneToN     is range 1      to 100000;
type zeroTo1    is range 0      to 1;
type oneTo1     is range 1      to 1;
type Integer    is range -100000 to 100000;
```

```
type Date       is array [1..9] of Char;
```

```
type StringSet  is Set of String;
type FacultySet is Set of Faculty;
type StudentSet is Set of Student;
type BicycleSet is Set of Bicycle;
```

Class GradClass is

```
  var program      : String;
  var year         : Integer;
  var graddate     : Date;
  var designator   : String is PrimaryKey;
```

```
  invariant        (size(this.MemberOf.Contains) >= 0 and
                    size(this.MemberOf.Contans) <= 1);
```

end Class;

Class Quarter is

```
  var qname        : String is PrimaryKey;
  var year         : Integer;
  var qstart       : Date;
  var qend         : Date;
```

```
  invariant        (size(this.Offering.Time) >= 0 and
                    size(this.Offering.Time) <= 100000);
```

end Class;

```

Class Course is
  var ctype          : String;
  var cnum           : String is PrimaryKey;
  var ctittle        : String;
  var cdesc          : String;
  var creditHours    : Integer;
  var lectureHours   : Integer;
  var abetDes        : Integer;
  var abetSci         : Integer;
  var abetMath       : Integer;
  var abetOther      : Integer;

  invariant          (size(this.Offering.Program) >= 0 and
                      size(this.Offering.Program) <= 100000);
end Class;

Class Section is
  var snumber        : Integer;
  var TaughtAs__Offering_Course_cnum : String is PrimaryKey
                                from Offering;
  var TaughtAs__Offering_Quarter_qname : String is PrimaryKey
                                from Offering;
  var Teaching__Faculty_Person_ssn    : String from Faculty;

  invariant          (size(this.TaughtAs.Actual)      >= 1 and
                      size(this.TaughtAs.Actual)      <= 100000)      and
                      (size(this.Assigned.AssignedTo) >= 0 and
                      size(this.Assigned.AssignedTo) <= 100000)      and
                      (size(this.Teaching.TaughtBy)   >= 1 and
                      size(this.Teaching.TaughtBy)   <= 1);
end Class;

Class Bicycle is
  var bikeSerialNum  : String is PrimaryKey;
  var Owns__Person_ssn : String from Person;

  invariant          (size(this.Owns.OwnedBy) >= 0 and
                      size(this.Owns.OwnedBy) <= 100000);
end Class;

```

```

Class Person is
  var lastname      : String;
  var firstname     : String;
  var initial       : String;
  var birthdate     : String;
  var ssn           : String is PrimaryKey;
  var height        : Integer;
  var weight        : Integer;

  invariant         (size(this.Owns.Owner) >= 0 and
                    size(this.Owns.Owner) <= 1);
end Class;

Class Faculty is Person with
  var academicRank   : String;
  var Person_ssn     : String is PrimaryKey from Person;

  invariant          (size(this.Teaching.Teaches) >= 1 and
                    size(this.Teaching.Teaches) <= 1)          and
                    (size(this.RAdvises.Advisor) >= 1 and
                    size(this.RAdvises.Advisor) <= 1);

  function getStudentsAdvised() : StudentSet is

    guarantees getStudentAdvised =
      { s | ( s : Student) s in this.RAdvises.Advisee };

  function getStudentsAdvisedSSN () : StringSet is

    guarantees getStudentsAdvisedSSN =
      { s | ( s : Student.ssn) s in this.RAdvises.Advisee };

  function getMyBikes () : BikeSet is

    guarantees getMyBikes =
      { b | ( b : Bicycle) b in this.Owns.OwnedBy };
end Class;

```

```

Class Student is Person with
  var gpa                      : Integer;
  var Person_ssn              : String is PrimaryKey
                              from Person;
  var MemberOf__GradClass_designator : String from GradClass;
  var RAdvises__Facutly_Person_ssn   : String from Faculty;

  invariant    (size(this.Assigned.Assignee) >= 0 and
                size(this.Assigned.Assignee) <= 1)          and
                (size(this.MemberOf.IsIn)    >= 0 and
                size(this.MemberOf.IsIn)    <= 100000)       and
                (size(this.RAdvises.Advisee) >= 0 and
                size(this.RAdvises.Advisee) <= 100000);

  function  getMyFaculty(InputQname : in String) : FacultySet is

    guarantees getMyFaculty(InputQname) =
      { f | ( f : Faculty, s : Section, o : Offering, q : Quarter )
            s in f.Teaching.Teaches and
            o in s.Taught_As.Theory and
            q in o.Offering.Time and
            q.qname = InputQname and
            this in o.Assigned.Assignee};

end Class;

AssocObject Offering is
  var snum          : Integer;
  var Course_cnum   : String is PrimaryKey from Course;
  var Quarter_qname : String is PrimaryKey from Quarter;

  invariant    (size(this.TaughtAs.Theory) >= 0 and
                size(this.TaughtAs.Theory) <= 1);

  role Program      : Course          is zeroToN;
  role Time         : Quarter         is zeroToN;
end AssocObject;

```

```

AssocObject AssignedAO is
  var Student_Person_ssn          : String is PrimaryKey
                                   from Student;
  var Section-TaughtAs__Offering_Course_cnum : String is PrimaryKey
                                   from Section;
  var Section-TaughtAs__Offering_Quarter_qname : String is PrimaryKey
                                   from Section;

  role Assignee      : Student      is oneTo1;
  role AssignedTo    : Section      is zeroToN;
end AssocObject;

```

```

Association Assigned is
  role Assignee      : Student      is oneTo1;
  role AssignedTo    : Section      is zeroToN;

  implemented by AssignedAO;
end Association;

```

```

Association MemberOf is
  role isIn          : Student      is zeroToN;
  role Contains      : GradClass    is zeroTo1;

  implemented by Student;
end Association;

```

```

Association RAdvised is
  role Advisor       : Faculty      is oneTo1;
  role Advisee       : Student      is zeroToN;

  implemented by Student;
end Association;

```

```

Association TaughtAs is
  role Actual        : Section      is oneToN;
  role Theory        : Offering     is zeroTo1;

  implemented by Section;
end Association;

```

```
Association Teaching is
  role Teaches      : Faculty      is oneTo1;
  role TaughtBy     : Section      is zeroToN;

  implemented by Section;
end Association;
```

```
Association Owns is
  role Owner        : Faculty      is zeroTo1;
  role OwnedBy      : Bicycle      is zeroToN;

  implemented by Bicycle;
end Association;
```

Appendix C. DDL Generated From School Specification

This appendix contains the DDL strings generated from the transformed specification in Appendix B. A table declaration statement represents each class and associative object. In its entirety, this database schema represents the relational persistent data storage structure for the OO original specification. The database defined below will act as the target for DML statements generated from the methods of the specification.

```
CREATE TABLE GradClass (
    program                VARCHAR2(25),
    year                   INTEGER,
    graddate                DATE,
    designator              VARCHAR2(25)    NOT NULL,
    PRIMARY KEY (designator))
```

```
CREATE TABLE Course (
    ctype                  VARCHAR2(25),
    cnum                   INTEGER          NOT NULL,
    ctitle                  VARCHAR2(25),
    cdesc                   VARCHAR2(25),
    creditHours             INTEGER,
    lectureHours            INTEGER,
    abetDes                  INTEGER,
    abetSci                  INTEGER,
    abetMath                 INTEGER,
    abetOther                INTEGER,
    PRIMARY KEY (cnum))
```

```
CREATE TABLE Person (
    lastname                VARCHAR2(25),
    midinitial              VARCHAR2(1),
    firstname               VARCHAR2(25),
    birthdate               DATE,
    ssn                     VARCHAR2(9)    NOT NULL,
    height                  INTEGER,
    weight                  INTEGER,
    PRIMARY KEY (ssn))
```



```

CREATE TABLE Bicycle (
    bikeSerialNum          VARCHAR2(25)    NOT NULL,
    Owns__Person_ssn       VARCHAR2(9),
    FOREIGN KEY (Owns__Person_ssn) REFERENCES Person(ssn),
    PRIMARY KEY (bikeSerialNum))

CREATE TABLE Faculty (
    academicRank           VARCHAR2(25),
    Person_ssn             VARCHAR2(9)    NOT NULL,
    FOREIGN KEY (Person_ssn) REFERENCES Person(ssn),
    PRIMARY KEY (Person_ssn))

CREATE TABLE Quarter (
    qname                  VARCHAR2(25)    NOT NULL,
    year                   INTEGER,
    qstart                 DATE,
    qend                   DATE,
    PRIMARY KEY (qname))

CREATE TABLE Student (
    gpa                    INTEGER,
    Person_ssn             VARCHAR2(9)    NOT NULL,
    MemberOf__GradClass_designator VARCHAR2(25),
    RAdvises__Faculty_Person_ssn VARCHAR2(9),
    FOREIGN KEY (Person_ssn) REFERENCES Person(ssn),
    FOREIGN KEY (MemberOf__GradClass_designator) REFERENCES GradClass(designator),
    FOREIGN KEY (RAdvises__Faculty_Person_ssn) REFERENCES Faculty(Person_ssn),
    PRIMARY KEY (Person_ssn))

CREATE TABLE Offering (
    code                   VARCHAR2(25),
    Course_cnum            INTEGER        NOT NULL,
    Quarter_qname          VARCHAR2(25)    NOT NULL,
    FOREIGN KEY (Course_cnum) REFERENCES Course(cnum),
    FOREIGN KEY (Quarter_qname) REFERENCES Quarter(qname),
    PRIMARY KEY (Course_cnum , Quarter_qname))

```

```

CREATE TABLE Section (
    snumber                      INTEGER,
    TaughtAs__Offering_Course_cnum  INTEGER      NOT NULL,
    TaughtAs__Offering_Quarter_qname VARCHAR2(25)  NOT NULL,
    Teaching__Faculty_Person_ssn    VARCHAR2(9),
FOREIGN KEY
    (TaughtAs__Offering_Course_cnum , TaughtAs__Offering_Quarter_qname)
    REFERENCES Offering(Course_cnum , Quarter_qname),
FOREIGN KEY (Teaching__Faculty_Person_ssn)
    REFERENCES Faculty(Person_ssn),
PRIMARY KEY
    (TaughtAs__Offering_Course_cnum , TaughtAs__Offering_Quarter_qname))

```

```

CREATE TABLE AssignedAO (
    Student_Person_ssn          VARCHAR2(9)      NOT NULL,
    Section_TaughtAs__Offering_Course_cnum  INTEGER      NOT NULL,
    Section_TaughtAs__Offering_Quarter_qname VARCHAR2(25)  NOT NULL,
FOREIGN KEY (Student_Person_ssn)
    REFERENCES Student(Person_ssn),
FOREIGN KEY
    (Section_TaughtAs__Offering_Course_cnum,
     Section_TaughtAs__Offering_Quarter_qname)
    REFERENCES
        Section(TaughtAs__Offering_Course_cnum,
                TaughtAs__Offering_Quarter_qname),
PRIMARY KEY
    (Student_Person_ssn , Section_TaughtAs__Offering_Course_cnum,
     Section_TaughtAs__Offering_Quarter_qname))

```

Appendix D. DML Generated From School Specification

This appendix contains a listing of methods from the original specification in Appendix A. After each method, a corresponding DML string is listed. The DML strings were generated by interpreting the post conditions of the methods and referencing the design decisions of the transformed specification shown in Appendix B. These DML statements are intended to be embedded in application code and applied to the database declared in Appendix C.

```
function getStudentsAdvised() : StudentSet is

    guarantees getStudentAdvised =
        {s | ( s : Student) s in this.RAdvises.Advisee };

SELECT Student.*, Person.*
FROM   Student, Person, Faculty
WHERE  this.Person_ssn = Faculty.Person_ssn           and
       Student.Person_ssn = Person.ssn                and
       Faculty.Person_ssn = Student.RAdvises__Faculty_Person_ssn

function getStudentsAdvisedSSN () : StringSet is

    guarantees getStudentsAdvisedSSN =
        { s | ( s : Student.ssn) s in this.RAdvises.Advisee };

SELECT Person.ssn
FROM   Student, Person, Faculty
WHERE  this.Person_ssn = Faculty.Person_ssn           and
       Student.Person_ssn = Person.ssn                and
       Faculty.Person_ssn = Student.RAdvises__Faculty_Person_ssn

function getMyBikes () : BikeSet is

    guarantees getMyBikes =
        { b | ( b : Bicycle) b in this.Owns.OwnedBy };

SELECT Bicycle.*
FROM   Bicycle, Faculty, Person
WHERE  this.Person_ssn = Faculty.Person_ssn           and
       Faculty.Person_ssn = Person.ssn                and
       Person.ssn       = Bicycle.Owns__Person_ssn
```

```

function  getMyFaculty(InputQname : in String) : FacultySet is

    guarantees getMyFaculty(InputQname) =
        { f | ( f : Faculty, s : Section, o : Offering, q : Quarter )
              s in f.Teaching.Teaches and
              o in s.Taught_As.Theory and
              q in o.Offering.Time      and
              q.qname = InputQname      and
              this in o.Assigned.Assignee};

SELECT Faculty.*, Person.*
FROM    Faculty, Person, Section, Offering, Quarter, Student, AssignedAO
WHERE this.Person_ssn      = Student.Person_ssn          and
      Faculty.Person_ssn   = Person.ssn                  and
      (Faculty.Person_ssn  = Section.Teaching__Faculty_Person_ssn  and
      (Offering.Course_cnum = Section.TaughtAs__Offering_Course_cnum and
      Offering.Quarter_qname = Section.TaughtAs__Offering_Quarter_qname and
      (Quarter.qname       = Offering.Quarter_qname          and
      ((InputQname         = Quarter.qname)                  and
      Student.Person_ssn   = AssignedAO.Assigned__Student_Person_ssn and
      Section.TaughtAs__Offering_Course_cnum =
          AssignedAO.Assigned__Section_TaughtAs__Offering_Course_cnum and
      Section.TaughtAs__Offering_Quarter_qname =
          AssignedAO.Assigned__Section_TaughtAs__Offering_Quarter_qname))))

```

Bibliography

1. Anderson, Gary L. *An Interactive Tool for Refining Software Specifications from a Formal Domain Model*. MS thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH, March 1999. DTIC Number: ADA361745.
2. Bourdeau, Robert H. and Betty G. C. Cheng. "A formal Semantics for Object Model Diagram," *IEEE Transactions on Software Engineering*, 21(10) (November 1995).
3. Cornn, Gary L. *A Common Object-Oriented Repository for a Software Synthesis System*. MS thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH, March 2000. AFIT/GCS/ENG/00M-05.
4. Corporation, Rational Software, "Rational Rose 98 Product Information," July 1999. <http://www.rational.com/products/rose/prodinfo.html>.
5. Date, C. J. *An Introduction to Database Systems* (sixth Edition). New York, New York: Addison-Wessley, 1995.
6. Fong, Joseph. "Converting Relational to Object-Oriented Databases," *SIGMOD Record*, 26(1) (March 1997).
7. Graham, Robert P. Jr. *Common Object-Oriented Imperative Language*. Air Force Institute of Technology, 1999.
8. Hodges, Julia and Shekar Ramanathan. "Extraction of Object-Oriented Structures from Existing Relational Databases," *SIGMOD Record*, 26(1) (March 1997).
9. Kissack, John A. *Transforming Aggregate Object-Oriented Formal Specifications to Code*. MS thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH, March 1999. DTIC Number: ADA361759.
10. Koch, George and Kevin Loney. *Oracle: The Complete Reference*. Berkley, California: McGraw-Hill, 1997.
11. Logic Works, Inc. *Getting Started Guide*, 1996.
12. Oracle, "Oracle Developer: Automating SQL Transformation," July 1999.
13. Pressman, Roger S. *Software Engineering: A Practitioner's Approach* (Fourth Edition). New York, New York: McGraw-Hill, 1997.
14. Products, Platinum, "SQL-Ease," July 1999.
15. Rational Software Corporation. *Rose 98 User Guide*, 1998.
16. Reasoning Systems Inc. *Refine User's Guide*, 1980.
17. Rumbaugh, James, et al. *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice-Hall, 1991.
18. Shanley, Linda. *Streamlining Query Generation. The Client Server*. Bently Systems, Inc., April 1998.

19. Spivey, J.M. *Understanding Z: A Specification Language and its Formal Semantics*. Cambridge: Cambridge University Press, 1988.
20. Tankersley, Travis W. *Generating Executable Code from Formal Specifications of Primitive Objects*. MS thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH, March 1999. DTIC Number: ADA361722.

Vita

Captain Steven R. Buckwalter was born on April 15, 1966 in Crossville, Tennessee. He graduated from Troy High School, Troy, Ohio, in 1984. He enlisted in the Air Force in August of 1984 and worked as a Nuclear Weapons Technician. During his enlisted career, 1984 through 1995, Captain Buckwalter was stationed at Blytheville AFB, Arkansas; Murted, Turkey; Hahn AB, Germany; Kleine Brogel, Belgium; and Kirtland AFB, New Mexico. He married Patricia Novak in June of 1989. In 1995 he graduated magna cum laude with a Bachelor of Science degree in Computer Science. In May of 1996, he was commissioned after completing Officer Training School. Steven and Patricia have two children, Benjamin and Daniel.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 074-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of the collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE 8 March 2000	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE Generating Executable Persistent Data Storage/Retrieval Code from Object-Oriented Specifications			5. FUNDING NUMBERS	
6. AUTHOR(S) Steven R. Buckwalter, Captain, USAF				
7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 P Street, Building 640 WPAFB OH 45433-7765			8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/00M-02	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFRL/IFTD Attn: Mr. Roy F. Stratton, Jr. 525 Brooks Rd. Rome, NY 13441-4505 (315) 330-3004			10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES Maj. Robert P. Graham, Jr. (937) 255-3636 x4595 Robert.Graham@afit.af.mil				
12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.			12b. DISTRIBUTION CODE	
ABSTRACT (Maximum 200 Words) This research creates a methodology and corresponding prototype for the transformation of object-oriented (OO) specifications to represent the corresponding relational schemas that are used to automatically generate database design language (DDL). The transformation design decisions and specifications are then used to generate database manipulation language (DML) that can be embedded within the software application code generated from the same OO specifications. This concept of developing a model for producing compilable and executable code from formal software specifications has long been a goal of software engineers. Previous research at the Air Force Institute of Technology (AFIT) has not focused on the representation of persistent data from OO software specifications. Relational databases are historically among the most popular methods of managing persistent data associated with software systems. However, there is not an automated tool available that will create the DDL and DML from OO specifications. This research develops a framework for combining these separate processes into a single step. Generating the relational database and the operations to manage data within the database from the formal software system specification. When combined with software system code generation, this research will allow the production of entire software systems to include the application code and persistent data management in a relational database.				
14. SUBJECT TERMS Software engineering, Transformation systems, Code generation, Relational database, DDL, DML			15. NUMBER OF PAGES 100	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	